



何永琪 主编 杨红涛 焦悦光 戴无惧 编写

嵌入式Linux

系统实用开发

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

本书从实用的角度出发,以 S3C2410A 芯片及 HY2410A 开发板为主,介绍了嵌入式 ARM 平台上 Linux 系统开发所需的各种技术,包括 ARM 处理器架构与汇编语言、嵌入式 Linux 开发环境的建立、C 及 C++ 语言要点、bootloader 和 Linux 内核的移植、Linux 系统应用编程、Linux 内核驱动编程和 Qt 图形界面应用的开发等,涵盖了嵌入式产品软件开发工作所需的各种主要技术。

本书是一本面向产品开发基本职业技能的嵌入式 Linux 软件开发入门书籍,适合于刚进入嵌入式行业的开发人员及技术爱好者阅读,也可供高等院校和各类职业教育院校信息技术相关专业人员作为参考资料。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

嵌入式 Linux 系统实用开发 / 何永琪主编. — 北京: 电子工业出版社, 2010.3
(实用为王)
ISBN 978-7-121-10039-0

I. 嵌… II. 何… III. Linux 操作系统—程序设计 IV. TP316.89

中国版本图书馆 CIP 数据核字(2009)第 224359 号

责任编辑: 董 英

印 刷:

装 订:

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱

邮编: 100036

开 本: 787×1092 1/16

印张: 46.5

字数: 1339 千字

印 次: 2010 年 3 月第 1 次印刷

定 价: 89.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zlts@phei.com.cn, 盗版侵权举报请发邮件到 dbqq@phei.com.cn。

服务热线: (010) 88258888。

前 言

嵌入式软件开发是嵌入式产品开发的关键技术之一，特别是基于 Linux 操作系统的嵌入式软件开发，已经成为嵌入式开发的主要领域，它不但涉及 Linux 操作系统内核的移植、剪裁和优化，还包含大量外设接口、总线等的驱动程序开发、移植和优化，以及大量的面向具体应用需求和产品要求的图形化应用软件开发。

本书是一本面向产品开发基本职业技能的嵌入式 Linux 软件开发入门书籍，适合于下列读者群：

- ◆ 嵌入式行业新入职的软硬件开发人员。
- ◆ 原来从事单片机系统开发，有意转向 SoC（片上系统）开发的技术人员。
- ◆ 高等院校和各类职业院校信息技术相关专业高年级学生和研究生一年级学生。
- ◆ 嵌入式技术爱好者。

本书也可作为嵌入式行业在职技术人员、高等院校和各类职业院校信息技术相关专业教师及实验辅导人员提供一定的技术参考。

本书结构

本书的结构安排反映了作者多年从事嵌入式软件开发的丰富经验和对嵌入式产品开发基本职业技能要求的深入理解，也参考了本书评审专家和友好读者的意见，同时考虑到了大多数读者的现实技术基础；当然，许多网络书店上大量读者对已出版的相关书籍所做的书评也给了作者非常有益的启示。本书整体结构实际上遵循了嵌入式产品软件开发的基本工作流程，从 Linux 操作系统下的基本编程和面向软件开发的 ARM 体系硬件平台关键特性及其编程应用，到 Linux 操作系统的内核编程及系统调用、内核移植与剪裁、驱动程序开发和移植，直至 Linux 环境下的图形化应用软件开发，涵盖了嵌入式产品软件开发工作的各种主要技术，并且最后通过一个实际项目开发综合再现了全书的核心内容。

总体内容

本书分为 6 个部分。

第 1 部分 嵌入式开发基础

第 1 章从产品、服务和应用的角度，概括地介绍嵌入式技术的主要市场应用、嵌入式产品开

发中涉及到的关键技术,让读者基本了解掌握嵌入式软件开发技术后能够从事哪些产品的开发、能够在哪些行业寻找适合的工作岗位。第 2 章从嵌入式系统软件产品开发的角度,重点阐述 C 语言编程的核心要领,特别是在 Linux 和嵌入式 Linux 环境下 C 语言编程的关键技术。第 3 章引导读者利用开源软件,搭建嵌入式 Linux 软件开发的基本工作环境,主要是帮助读者在目前最流行的 Linux 发行版本——Debian 5.0 上顺利搭建一套实用性软件开发环境。

第 2 部分 ARM 架构与编程

这一部分包括第 4 章到第 7 章,主要是与硬件平台和 ARM CPU 基础指令有关的内容。第 4 章主要从软件开发编程的角度,详细介绍 ARM9 体系的核心架构、ARM 常用指令及其编程应用和 ARM 汇编语言程序开发方法。第 5 章主要是引导读者顺利完成一套嵌入式系统的固件(Firmware)开发环境的搭建和使用。第 6 章首先介绍目前市场上比较常见的三星 S3C2410A 嵌入式处理器的基本体系结构、工作原理,然后详细分析 S3C2410A 处理器主要接口和外设控制器的工作原理以及使用方式。第 7 章以目前嵌入式 Linux 系统中最常见的系统引导(bootloader)软件 U-boot 为例,帮助读者理解 bootloader 的工作机制、开发移植流程和系统应用。

第 3 部分 Linux 系统编程

这一部分包括第 8 章到第 12 章,是 Linux 软件开发的基础。第 8 章是 Linux 系统编程的基础,重点讨论 Linux 系统中程序的层次结构和内存映像、Linux 库函数的错误处理以及环境变量等。第 9 章主要讨论 Linux 的文件系统及其编程应用,由于 Linux 系统中很多设备的操作是以文件系统的方式处理的,因此文件系统在 Linux 中十分重要。第 10 章详细介绍操作系统的—个核心概念——进程,还涉及到进程的同步和进程间通信等基本编程应用。第 11 章讲解 socket 编程这一 Linux 网络通信应用的基础。第 12 章主要讲解多线程程序的编程开发。

第 4 部分 内核与驱动编程

这一部分包括第 13 章到第 16 章。第 13 章介绍实模式与保护模式、用户态与内核态等基本概念,同时还介绍了内核编程的一些特点,以一个简单的字符设备为例说明了驱动编程的一般方法。第 14 章主要讨论 Linux 编程的内核接口及其应用。第 15 章介绍 Linux 2.6 版本的设备模型和编程应用要点。第 16 章以输入设备驱动和 USB 设备驱动为例介绍 Linux 设备驱动程序开发的基本原理和方法。由于很多读者没有编程经验,或者以前主要从事应用编程,而从应用编程到内核和驱动编程无论是概念还是方法都需要一定的转变,这一部分的目的就是引导读者完成这一转变。

第 5 部分 嵌入式 Linux 系统构建

这一部分内容相对简单,作者认为绝大多数读者不会从事 Linux 内核的开发,而是有目的地使用或者剪裁稳定版本的 Linux 内核,因此在第 17 章中主要介绍 Linux 内核的配置原理与方法,并举例说明其中关键操作的基本流程,在第 18 章中则主要讲解并演示 Linux 根文件系统构建和移植的方法。通过这两章,读者基本可以掌握嵌入式 Linux 系统内核的配置方法,进行简单的内核移植,以及按应用要求构造根文件系统。

第 6 部分 应用编程

这一部分包括第 19 章到第 22 章，主要目的是帮助读者熟悉针对诸如智能手机、自助终端等终端设备类功能需求的应用开发。第 19 章从开发编程的角度介绍 C++ 特有的概念和用法，适合那些对 C 语言有一定基础的读者快速进入 C++ 编程领域。第 20 章以目前嵌入式行业最常见的开发环境——Qt 4.5.2 的开源版本为例，介绍如何使用 Qt 平台进行嵌入式 GUI 的开发。第 21 章则简单介绍最常用的嵌入式数据库 SQLite 的应用开发方法。第 22 章通过一个实际产品开发中的典型项目案例，将本书核心内容贯穿于一体，使读者有机会全方位地了解嵌入式产品的软件开发流程和实施方法。

主要特色

本书的作者长期从事嵌入式软件开发工作，本书既是他们经验和教训的汇总，也是他们对嵌入式产品软件开发的理解。书中内容取舍与结构安排均面向嵌入式软件开发的实际需要；书中的代码均经过编译和运行，进行了严格的测试，部分代码来自实际产品开发工作并且经过终端产品的长期应用考验；Linux 内核版本选择的是初稿完成时的最新内核稳定版本——Linux 2.6.30；应用开发环境则选择的是最新的跨操作系统开发平台——Qt Embedded 4.5.1，特别是本书支持网站上提供的 Qt Embedded 4.5.1 是本书初稿完成时全球唯一的支持简体中文处理的版本（这个版本也是本书几位作者移植成功的）。

本书的支持网站是 <http://www.cjhytec.com>，提供例程源码的下载。

本书作者

本书由何永琪教授主编，杨红涛负责编写第 4 章到第 12 章、第 17 章、第 18 章、第 21 章和第 22 章，焦悦光负责编写第 2 章、第 3 章、第 13 章到第 16 章、第 19 章和第 20 章，戴无惧负责编写前言和第 1 章并起草全书大纲。邓莹莹为第 19 章和第 20 章提供部分章节初稿，袁裕芳、张欣然和朱子豪为第 20 章、第 21 章和第 22 章提供部分章节初稿，周聪、林胜朋和王松为第 6 章和第 22 章提供部分章节初稿，吴龙和李英良为第 11 章、第 15 章和第 16 章提供部分章节初稿，魏大庆和江中舟负责全书图例绘制。焦悦光最后统稿全书。

序

20 世纪 90 年代以来,计算机技术、通信技术和集成电路技术飞速发展,并且相互融合,导致了嵌入式技术及其应用的产生和迅猛发展,对全球包括我国的工业、农业、科技、军事、教育、文化等领域产生了深远的影响,也有力地促进了全球化和信息化。目前,嵌入式技术已经成为信息产业发展的核心领域之一,对国民经济、国防安全以及人们的日常生活、工作和学习等都发挥着日益重要的作用。

随着改革开放步伐的加大,特别是加入了 WTO 之后,我国已成为全球电子设备和通信终端产品的主要生产制造基地之一。2008 年,全球 52% 的手机、90% 以上的笔记本电脑,特别是 iPhone,都是在中国大陆生产的,而多媒体播放器、数字电视机顶盒、无线终端、电子词典、银行 ATM 机、商业 POS 机、智能家电、工业控制器、传感器等嵌入式产品的主要生产制造基地也都在中国。

2009 年 1 月 7 日,我国政府向三个运营商发放了 3G 牌照,由大唐电信集团代表我国提出的拥有自主知识产权的 TD-SCDMA 3G 国际标准由全球最大的移动运营商——中国移动主导正式开始商用,这标志着我国进入了 3G 时代。移动互联网应用是 3G 时代的发展主流,消费电子、计算机与无线移动通信进一步融合,出现了智能手机、3G 上网本、3G 电子书等,而数码相机、数码摄像机等也将具有 3G 无线通信功能。2009 年全球移动通信用户数预计将超过 46 亿,中国及新兴市场仍是用户数增长的主要驱动力量。未来人与人之间的通信将逐步趋于饱和,而物与物、人与物之间的通信发展潜力将更为巨大。据预测,到 2020 年全球将有 500 亿个互联终端,而这些终端都需要嵌入式技术来实现。物联网、传感网都需要采用嵌入式技术来实现低功耗、低成本,以及恶劣环境下使用等问题。2009 年 8 月 7 日,温家宝总理在无锡考察时提出了“感知中国”,要求把传感系统和 3G 中的 TD-SCDMA 技术结合起来。随着移动通信网在我国的广泛覆盖,物联网、传感网已成为我国新兴战略产业之一,而这一切都离不开嵌入式技术,也将促进我国的信息化和工业化融合(即“两化融合”),实现经济结构调整、产业转型和升级。可见,未来嵌入式技术及其应用将十分广泛,战略意义也十分重大。

因此,我国嵌入式系统技术人才的需求将越来越大。但是,与嵌入式系统产业的蒸蒸日上相比,我国嵌入式系统技术教育和人才培养相对落后,导致嵌入式系统技术人才的培养和成长不能满足嵌入式系统产业快速发展的需求,具备独立从事嵌入式系统产品开发的专业人才尤为缺乏。

为了支撑嵌入式系统技术的人才培养,便于需要掌握嵌入式系统技术人员的学习,本书以传授实用性知识、培养实际开发技能为宗旨,针对嵌入式系统产品开发的基本技能要求,凝聚几位长期从事嵌入式产品开发专家十余年的开发经验,并融合最新技术成果,编写而成,这在业界是不多见的。

本书在嵌入式系统技术知识与实际产品开发技能之间的“鸿沟”上构建了很好的桥梁，相信嵌入式系统技术初学者和大、专院校电子信息类学生以及嵌入式系统产品开发人员，将能够通过本书更加容易、透彻地了解和理解嵌入式系统基本技术，并借以提高实际产品开发技能，快速有效地走进嵌入式系统产品开发领域。

大唐电信科技产业集团总工程师
国家 863 信息技术领域专家组成员

A handwritten signature in black ink, appearing to read '陈震' (Chen Zhen).

2009.12.1

主 编 致 辞

近十年来,计算机技术、通信技术和微电子技术得到迅猛发展,对我国和世界的工业、农业、军事、科技、教育、文化等领域产生了长期而深刻的影响。这三大技术的相互融合已成必然趋势,导致基于定制化的嵌入式技术得到快速发展和普及应用,使之成为信息产业发展的核心领域之一,在工业控制、农业自动化、通信、交通、金融、商业、军事、航空、航天、广电、医疗、家用电子等各个领域发挥日益重要的作用。

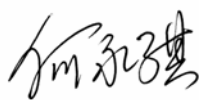
嵌入式技术作为当今最有发展前途的信息产业应用技术之一,催生了巨大的产业需求和人才需求。我国已成为全球主要信息产业设备和终端产品的主要生产国和出口国,基于嵌入式技术的低成本与定制化特点及其广泛的应用领域和广阔的应用前景,嵌入式技术规模产业已经形成,并得到迅速发展,将逐步形成以大中型研发生产企业为群体的、应用门类齐全的、巨大的嵌入式产品产业群。同时,由于嵌入式技术的定制化特点,所有产品均需要根据其应用特点进行定制化的设计和开发,从而导致嵌入式从业技术人员的规模不断扩大,使嵌入式技术产业成为最具发展力的、人心向往的朝阳产业之一。

随着嵌入式技术的快速发展和嵌入式产业的不断扩大,形成了对嵌入式技术人才的大量需求,特别是具备独立研发能力的嵌入式产品开发人才。但是,由于我国高等教育体制严重滞后、教育与产业结合严重脱节,很难实现对软硬件技术兼备的嵌入式技术研发人员的有效培养,形成了重知识、轻技能,重软件、轻硬件的人才培养怪圈,加剧了嵌入式技术实用性人才的匮乏。

为了使大专院校电子信息类的大专生、本科生和研究生能够更加顺利地进入嵌入式技术领域,也为了使广大的嵌入式产品研发人员能够真正理解嵌入式技术研发的真谛,我们以实用性开发技能和实际产品开发为导向,组织长期从事嵌入式产品开发工作的资深研发人员,结合十多年产品开发经验和嵌入式技术最新发展成果,写成此书。本书力图弥补学院式教学的缺陷,在知识和技能之间架接通畅的桥梁,使读者顺利实现从掌握知识到掌握开发技能的转变,深刻理解嵌入式产品开发的真谛。为此,本书注重嵌入式技术开发的结构性和实用性,力图让读者掌握嵌入式技术开发的工作流程、工作环境、工作方法以及关键技术的实现,并配以取材于实际产品开发的典型案例,使读者能够直接探索嵌入式技术开发的奥秘。

在本书成书过程中,得到了北京诚捷鸿远通信技术有限公司及其产品研发人员的大力支持,在此特向他们表示诚挚的谢意。

由于水平有限,书中不足甚至错误之处在所难免,欢迎广大读者批评和指正。



2009年11月于北京大学

编 委 会

主 编：何永琪 北京大学 教授、博士生导师

编 委：（按姓氏笔画顺序）

万国龙 北京航空航天大学 教授、博士生导师

严蔚敏 清华大学 教授、博士生导师

野永东 中国电信北京研究院 教授级高级工程师

余 综 华北计算技术研究所 研究员

张鲁生 台湾 Linux 学会 荣誉会长

编写者简介

杨红涛：1991 年获得厦门大学工学学士学位，1998 年获得电信科学技术研究院通信系统与网络专业工学硕士学位。先后担任过多家公司的项目负责人和技术总监，并曾在电信科学技术研究院(大唐电信)作为软件子项目负责人承担国家 863 信息领域重大专项的研发工作。

焦悦光：1995 年考入清华大学电子工程系，2004 年获得清华大学电子工程系工学博士学位。参加过多个国家 863 信息领域重要项目，负责软件开发和调试工作，并曾在联想研究院从事超宽带无线通信和 USB 显示系统的开发工作。

戴无惧：1989 年获得西安电子科技大学工学学士学位，1998 年获得西安电子科技大学工学硕士学位，2002 年获得清华大学工学博士学位。先后参加过国家七五项目、国家八五项目、邮电部八五项目、国家 863 信息领域重要项目，负责开发相关工作。

本书使用说明

一、关于本书程序代码验证用的嵌入式系统开发板平台

本书所有程序代码均依托北京诚捷鸿远通信技术有限公司提供的嵌入式系统系列开发板平台进行开发与调试，并通过了系统验证。下面列出了所用开发板的技术配置参数。

HY2410A 开发板平台

- ◆ 处理器：三星 S3C2410A 嵌入式处理器，主频 200MHz，ARM920T 内核
- ◆ 64MByte SDRAM（可扩展到 128MByte）
- ◆ 64MByte NAND Flash
- ◆ 2 个 USB 1.1 Host 接口
- ◆ 1 路立体声音频输出
- ◆ 1 路立体声音频输入
- ◆ 1 个 RS-232C 串口
- ◆ 1 个 SD/MMC 读卡器
- ◆ 1 个 10Mbit/s 以太网接口
- ◆ 1 个 20 针 ARM JTAG 接口



HY2410A

HY9315E/HY9307E 开发板平台

- ◆ 处理器：CIRRUS LOGIC EP9315 或 EP9307，主频 200MHz，ARM920T 内核
- ◆ 协处理器：Maverick Crunch 数学协处理器，MaverickKey 加密处理器，2D 硬件图形加速器
- ◆ 64MByte SDRAM（可扩展到 128MByte）
- ◆ 32MByte NOR Flash（可扩展到 64MByte）
- ◆ 1 个 IDE 接口（仅 HY9315E 支持）
- ◆ 10/100Mbit/s 自适应以太网接口
- ◆ 3 个 UART 接口和 1 个红外接口
- ◆ 6 个 USB 2.0 全速接口（支持 OHCI）
- ◆ 1 个 20 针 ARM JTAG 接口
- ◆ 双通道串行音频接口（AC'97）
- ◆ 1 路 LCD 和 1 路 VGA 输出（可同时使用）



HY9315E

HY9263E 开发板平台

- ◆ 处理器：Atmel AT91SAM9315，主频 200MHz，ARM926EJ-S 内核，集成 2D 硬件图形加速器
- ◆ 64MByte SDRAM（可扩展到 128MByte）和 4MByte PSRAM
- ◆ 256MByte NAND Flash（可扩展到 1GByte）
- ◆ 1 个 IDE 接口和 1 个摄像头接口
- ◆ 1 个 10/100Mbit/s 自适应以太网接口
- ◆ 3 个 UART 接口（RS-232C、RS-485 和调试串口各一个）和 2 个 SD/MMC 卡接口
- ◆ 5 个 USB 2.0 全速接口和 1 个 USB 2.0 高速 Device 接口
- ◆ 1 个 CAN 总线接口(兼容 Part 2.0A 和 Part 2.0B)
- ◆ 1 个 6 通道音频接口（支持 AC'97）
- ◆ 1 个 20 针 ARM JTAG 接口
- ◆ 1 路 LCD 输出



HY9263E

HYL137E 开发板平台

- ◆ 处理器：TI OMAP-L137，同时集成 ARM926EJ-S 内核和 TI C674x DSP 核，主频均为 300MHz
- ◆ 64MByte SDRAM（可扩展到 128MByte）
- ◆ 256MByte NAND Flash（可扩展到 1GByte）
- ◆ 10/100Mbit/s 自适应以太网接口
- ◆ 1 个 RS-232C 和 1 个 SD/MMC 卡接口
- ◆ 4 个 USB 2.0 全速接口和 1 个 USB 2.0 OTG 高速接口
- ◆ 1 组立体声音频接口
- ◆ 1 个 20 针 ARM JTAG 接口
- ◆ 1 路 LCD 输出

HY6410S 开发板

- ◆ 处理器：三星 S3C6410A 嵌入式处理器，主频 533MHz，ARM 1176JZF-S 内核，集成 2D/3D 图形加速器
- ◆ 64MByte MobileDDR（可扩展到 512MByte）
- ◆ 256MByte NAND Flash（可扩展到 2GByte）
- ◆ 4 个 USB 1.1 Host 接口（12Mbit/s）和 1 个 USB 2.0 OTG 高速接口
- ◆ 1 个摄像头接口和 1 组立体声音频输出/输入（支持 AC'97 和 PCM）
- ◆ 1 个 RS-232C 接口和 1 个 SD/MMC 读卡器接口
- ◆ 1 个 10Mbit/s 以太网接口
- ◆ 1 个 20 针 ARM JTAG 接口

二、关于本书的程序代码下载

本书所有程序代码均由北京诚捷鸿远通信技术有限公司开发或验证。



程序代码下载地址：<http://www.cjhytec.com>。

三、关于本书的疑难问题、勘误和技术服务

- ◆ 如读者阅读时，存在疑难问题或其他技术问题，欢迎读者及时提出，本书作者将尽快予以回复。
- ◆ 如读者阅读时，发现本书的错误，敬请提出，本书作者将期待与读者进行认真的沟通与讨论。
- ◆ 本书作者希望能够就各种技术问题与读者进行认真沟通与讨论，并接受相关问题咨询和技术咨询。
- ◆ 沟通与讨论可以通过电子邮件方式（电子邮件地址：cjhy@cjhytec.com），也可以登录北京诚捷鸿远通信技术有限公司网站（网站地址：<http://www.cjhytec.com>）技术论坛。

四、关于本书推荐的嵌入式系统系列开发板平台订购

本书推荐读者使用上述嵌入式系统开发板平台。

- ◆ 有关上述嵌入式系统系列开发板平台的详细配置和技术参数，可登录北京诚捷鸿远通信技术有限公司网站 <http://www.cjhytec.com> 进行了解。
- ◆ 本书承诺以优惠价格提供上述嵌入式系统系列开发板平台，并可通过北京诚捷鸿远通信技术有限公司网站 <http://www.cjhytec.com> 或电子邮件 cjhy@cjhytec.com 订购。



目 录

第 1 部分 嵌入式开发基础.....	1
第 1 章 引言	2
1.1 嵌入式产业概况	2
1.2 Linux 操作系统.....	5
1.3 ARM 体系概况	8
1.3.1 ARM7	10
1.3.2 ARM9	11
1.3.3 ARM9E	11
1.3.4 ARM10E	12
1.3.5 ARM11	12
1.3.6 Cortex	12
1.4 常见嵌入式产品及其基本平台简介	13
1.4.1 学习开发板.....	13
1.4.2 行业终端.....	16
1.4.3 工业控制.....	18
1.4.4 手持娱乐.....	19
1.4.5 医疗仪器.....	21
1.4.6 汽车电子.....	22
1.4.7 智能本	23
1.5 嵌入式产品开发基本流程.....	25
第 2 章 C 语言编程要点	28
2.1 数据类型.....	28
2.2 常数	29
2.3 变量	31
2.3.1 变量的定义与初始化	31
2.3.2 变量的访问.....	31
2.3.3 左值与右值.....	31
2.3.4 只读变量.....	32
2.4 操作符.....	32
2.4.1 只读操作符.....	32
2.4.2 读写操作符.....	34
2.4.3 类型转换操作符.....	35
2.4.4 sizeof 操作符	36
2.5 表达式和语句	36
2.5.1 表达式	36



2.5.2 语句	37
2.6 复合类型	38
2.6.1 数组	38
2.6.2 结构体	40
2.6.3 位域	42
2.6.4 数据的对齐	43
2.6.5 联合体	44
2.6.6 类型嵌套	44
2.6.7 类型别名	45
2.6.8 枚举类型	46
2.7 流程控制	46
2.7.1 顺序结构	46
2.7.2 分支结构	47
2.7.3 隐含的分支结构	49
2.7.4 循环结构	49
2.7.5 goto	52
2.8 函数	52
2.8.1 声明与定义	52
2.8.2 函数的调用与传值方式	54
2.8.3 函数与复合类型	55
2.8.4 内联函数	56
2.8.5 变量的作用范围与生存期	56
2.9 指针	59
2.9.1 指针与变量	59
2.9.2 指针与操作符	62
2.9.3 指针与数组	63
2.9.4 字符串	64
2.9.5 指针与结构体	65
2.9.6 指针与函数	66
2.9.7 回调函数	69
2.9.8 函数指针类型转换	70
2.10 预处理语句	71
2.10.1 文件包含	71
2.10.2 宏定义	71
2.10.3 宏与函数	73
2.10.4 代码分支	74
第 3 章 开发环境	76
3.1 Linux 使用基础	76
3.1.1 命令参数与选项	77
3.1.2 文件、目录和路径	77
3.1.3 用户与权限	78
3.1.4 硬链接与符号链接	78
3.1.5 命令使用技巧	79
3.2 Linux 常用命令	79



3.2.1	查阅手册	80
3.2.2	文件相关	81
3.2.3	文件内容相关	84
3.2.4	压缩与解压缩	86
3.2.5	文件系统与磁盘	88
3.2.6	用户与权限	89
3.2.7	进程管理	91
3.2.8	系统信息	92
3.2.9	网络	93
3.3	Shell 使用进阶	96
3.3.1	重定向	96
3.3.2	管道	98
3.3.3	变量与替换	98
3.3.4	环境变量	100
3.3.5	脚本	101
3.3.6	脚本编程	102
3.3.7	作业管理	103
3.4	Debian 5.0 的安装与使用	104
3.4.1	安装 Debian 5.0	105
3.4.2	Debian 5.0 的基本操作	105
3.4.3	常用软件的安装与使用	108
3.4.4	从源码安装软件	113
3.4.5	安装编译环境	115
3.5	建立交叉编译环境	117
3.5.1	下载安装	117
3.5.2	从源码编译安装	117
3.6	vi 编辑器	121
3.6.1	vi 的工作模式	121
3.6.2	普通模式	122
3.6.3	命令行模式	123
3.6.4	寄存器	124
3.6.5	与编程有关的技巧	124
3.7	gcc 工具链	124
3.7.1	编译过程	124
3.7.2	gcc 用法	125
3.8	make 与 Makefile	127
3.8.1	make 工具的使用	127
3.8.2	Makefile	128
3.9	gdb 调试工具	130
3.9.1	调试本地程序	130
3.9.2	远程调试	131
3.10	buildroot 开发工具	132
第 2 部分	ARM 架构与编程	135
第 4 章	ARM 处理器架构与编程模型	136
4.1	嵌入式硬件系统	137



4.1.1	嵌入式系统架构	137
4.1.2	S3C2410A 地址映射	138
4.1.3	HY2410A 开发板硬件配置	139
4.2	ARM 架构概述	139
4.2.1	ARM 处理器模式与寄存器组	139
4.2.2	ARM 异常与异常向量表	142
4.2.3	程序状态寄存器	143
4.2.4	大端与小端存储格式	145
4.3	ARM 指令集概述	146
4.3.1	ARM 汇编指令格式	146
4.3.2	数据处理指令	148
4.3.3	存储器访问指令	150
4.3.4	分支指令	152
4.3.5	软中断指令	153
4.3.6	程序状态寄存器传送指令	154
4.3.7	乘法指令	154
4.3.8	协处理器指令	155
4.3.9	伪指令	157
4.4	GNU ARM 汇编	158
4.4.1	基本语法	158
4.4.2	GNU ARM 汇编伪指令	159
4.5	汇编与 C 语言	163
4.5.1	程序及其二进制映像	163
4.5.2	程序的编译与运行	164
4.5.3	ATPCS 约定	165
4.5.4	汇编与 C 语言的对照	167
4.5.5	函数调用与栈	172
4.5.6	堆的概念	175
4.6	汇编与 C 语言混合编程	175
4.6.1	C 语言调用汇编函数	175
4.6.2	汇编语言中使用 C 全局变量	176
4.6.3	内嵌汇编	177
第 5 章	搭建嵌入式固件开发平台	181
5.1	硬件设备与软件环境	181
5.2	搭建开发环境	182
5.2.1	硬件连接	182
5.2.2	使用终端软件	183
5.2.3	下载和执行程序	184
5.3	创建固件程序	186
第 6 章	S3C2410 接口与编程	189
6.1	软中断异常编程	189
6.1.1	软中断异常入口	189
6.1.2	软中断异常应用例程	190
6.2	中断控制器及外部中断编程	193



6.2.1	中断体系结构	193
6.2.2	中断控制器	194
6.2.3	中断源安排	194
6.2.4	中断控制器寄存器配置	196
6.2.5	中断应用例程	197
6.3	定时器及其编程	201
6.3.1	定时器体系	201
6.3.2	定时器单元工作原理	202
6.3.3	定时器寄存器配置	203
6.3.4	定时器应用例程	205
6.4	GPIO 接口	209
6.5	UART 控制器及串口通信应用	211
6.5.1	UART 通信原理	211
6.5.2	RS-232C 串行接口标准	212
6.5.3	UART 控制寄存器	213
6.5.4	串口通信应用例程	217
6.6	NAND Flash 芯片与控制器	219
6.6.1	NAND Flash 的访问	220
6.6.2	NAND Flash 控制器	222
6.6.3	NAND Flash 控制器编程实例	223
第 7 章	U-boot 源码分析与移植	227
7.1	bootloader 的概念	227
7.1.1	bootloader 的启动过程	228
7.1.2	bootloader 的操作模式	228
7.1.3	ARM bootloader 的特点	229
7.2	使用 U-boot	230
7.2.1	U-boot 主要命令与环境变量	230
7.2.2	使用实例	232
7.3	U-boot 源码分析	233
7.3.1	总体架构与内存布局	233
7.3.2	源码目录	234
7.3.3	实现分析	235
7.4	U-boot 移植	244
7.4.1	源码修改	244
7.4.2	配置和编译	257
第 3 部分	Linux 系统编程	259
第 8 章	Linux 系统编程基础	260
8.1	系统调用与 API	260
8.2	程序的生成与执行	262
8.3	API 的错误处理	263
8.4	命令行参数与环境变量	263



第 9 章 Linux 文件系统编程	266
9.1 文件的概念	266
9.2 文件描述符与索引节点	267
9.3 文件操作的系统调用接口	268
9.3.1 打开文件	268
9.3.2 从文件读取数据	270
9.3.3 写数据到文件	272
9.3.4 发送控制命令	273
9.3.5 关闭文件	274
9.4 标准 I/O 函数库	275
9.4.1 fopen	276
9.4.2 fread 和 fwrite	276
9.4.3 fclose	277
9.4.4 fflush	277
9.4.5 fseek 和 ftell	277
9.4.6 fgetc, getc 和 getchar	278
9.4.7 fputc, putc 和 putchar	278
9.4.8 fgets 和 gets	279
9.4.9 fputs 和 puts	280
9.4.10 fprintf, printf 和 sprintf	280
9.4.11 fscanf, scanf 和 sscanf	282
9.4.12 标准 I/O 错误处理	284
第 10 章 深入理解进程	285
10.1 Linux 中的进程	285
10.1.1 创建进程	286
10.1.2 执行程序	287
10.1.3 进程的内存布局	288
10.1.4 进程的状态迁移	290
10.1.5 进程的终止	290
10.2 进程与信号	291
10.2.1 Linux 中的信号处理机制	292
10.2.2 发送信号	293
10.2.3 捕捉信号	295
10.2.4 屏蔽信号	299
10.2.5 信号安全函数	301
10.3 进程与文件	302
10.3.1 内核文件管理	302
10.3.2 进程中的文件	303
10.3.3 文件的重定向	306
10.3.4 文件控制	307
10.4 进程间通信	308
10.4.1 Linux 中的 IPC	309
10.4.2 信号灯与进程同步	310
10.4.3 管道	313



10.4.4	命名管道	314
10.4.5	共享内存	317
10.4.6	消息队列	321
第 11 章	socket 编程	326
11.1	网络协议层次模型	326
11.2	socket 编程接口	327
11.2.1	打开 socket	327
11.2.2	socket 地址	329
11.2.3	网络字节序	330
11.2.4	socket 与地址的绑定	331
11.2.5	侦听	331
11.2.6	接受连接请求	332
11.2.7	连接	332
11.2.8	关闭和切断连接	333
11.2.9	发送数据	333
11.2.10	接收数据	334
11.2.11	使用 socket 选项	335
11.2.12	阻塞与非阻塞操作	337
11.2.13	可靠的发送与接收操作	338
11.2.14	多路复用	339
11.3	socket 编程实例	340
11.3.1	TCP 与 UDP 程序流程	340
11.3.2	TCP 通信例程	340
11.3.3	多进程并发服务器应用	343
11.3.4	多路复用服务器应用	349
11.3.5	UDP 服务器应用	354
第 12 章	多线程并发程序设计	358
12.1	线程的概念	358
12.2	线程编程接口	358
12.2.1	线程的创建与退出	359
12.2.2	线程属性	359
12.2.3	线程的分离状态	361
12.2.4	线程应用实例	361
12.3	线程的同步	362
12.3.1	使用信号灯进行线程同步	363
12.3.2	使用互斥体	368
12.3.3	使用条件变量	371
12.4	多线程并发程序设计	375
12.4.1	多线程并发服务器应用	375
12.4.2	消费者/生产者模型	381
12.4.3	线程池应用	382

第 4 部分 内核与驱动编程	389
第 13 章 内核编程初步	390
13.1 从用户态到内核态	390
13.1.1 实模式与保护模式	390
13.1.2 用户态与内核态	391
13.1.3 内核编程的特点	391
13.1.4 内核模块与驱动	392
13.2 内核模块编程	393
13.2.1 编写源码	393
13.2.2 <code>printk</code> 函数	394
13.2.3 编译内核模块	396
13.2.4 加载与卸载	397
13.2.5 模块参数	397
13.3 字符设备驱动	399
13.3.1 设备文件与设备号	399
13.3.2 字符设备编程接口	400
13.3.3 文件操作	404
13.3.4 访问用户态内存	408
13.3.5 动态分配内存	409
13.3.6 内存操作	409
13.3.7 字符设备驱动例程	415
第 14 章 内核编程接口	421
14.1 双向环形链表	421
14.1.1 定义与初始化	421
14.1.2 链表操作	422
14.1.3 链表的使用	424
14.2 等待与延时	426
14.2.1 调度与抢占	426
14.2.2 进程运行状态	428
14.2.3 定时	429
14.2.4 等待队列	429
14.2.5 阻塞与非阻塞操作	432
14.2.6 延时	433
14.2.7 等待队列例程	434
14.3 定时器与延期工作	440
14.3.1 定时器	440
14.3.2 <code>tasklet</code>	444
14.3.3 工作队列	446
14.3.4 定时器例程	451
14.4 自旋锁与同步	455
14.4.1 并发与竞态	455
14.4.2 自旋锁	456
14.4.3 原子上下文	460

14.4.4	读写锁	462
14.4.5	原子类型	464
14.4.6	比特位操作	467
14.4.7	互斥体	468
14.4.8	信号灯	470
14.4.9	读写信号灯	472
14.5	端口 IO 和内存映射 IO	473
14.5.1	端口 IO	474
14.5.2	内存映射 IO	477
14.5.3	内存屏障	482
14.6	中断	483
14.6.1	申请和释放中断	483
14.6.2	中断处理函数	484
14.6.3	中断的禁止和使能	485
14.6.4	线程化中断	486
14.6.5	共享中断	486
第 15 章	Linux 2.6 设备模型	488
15.1	对象与集合	488
15.1.1	引用计数	488
15.1.2	内核对象	489
15.1.3	内核对象的类型	492
15.1.4	内核集合	496
15.1.5	内核集合与对象例程	500
15.2	设备管理	504
15.2.1	设备	504
15.2.2	错误码与指针	507
15.2.3	驱动	507
15.2.4	总线	508
15.2.5	类别	510
15.2.6	接口	512
15.3	常见总线与类别	513
15.3.1	platform 总线	513
15.3.2	misc 类别	516
第 16 章	Linux 驱动实例详解	518
16.1	输入设备驱动	518
16.1.1	输入设备编程接口	519
16.1.2	触摸屏驱动例程	523
16.2	USB 驱动	532
16.2.1	USB 概述	533
16.2.2	USB 驱动模型	534
16.2.3	USB 驱动编程接口	535
16.2.4	USB 接口与端点	537
16.2.5	USB 类别	540
16.2.6	URB	541



16.2.7	同步传输接口	546
16.2.8	USB 锚	547
16.2.9	USB 驱动范例分析	549
第 5 部分	嵌入式 Linux 系统构建	559
第 17 章	Linux 内核构建	560
17.1	内核编译过程	560
17.2	内核配置系统架构	562
17.2.1	内核 Makefile	562
17.2.2	KBuild 配置系统	566
17.3	增加代码到内核	572
17.4	内核配置简介	573
17.5	启动内核	575
第 18 章	根文件系统构建	576
18.1	init 进程	576
18.2	创建根文件系统	578
18.2.1	创建目录	578
18.2.2	创建设备文件	579
18.2.3	安装共享库	579
18.2.4	安装 busybox	580
18.2.5	创建配置文件	582
18.3	挂载根文件系统	583
18.3.1	使用网络文件系统	583
18.3.2	使用 Flash 文件系统	584
第 6 部分	应用编程	587
第 19 章	C++ 语言编程要点	588
19.1	布尔型数据	588
19.2	引用	588
19.3	类和对象	590
19.3.1	类和对象的定义	590
19.3.2	构造与析构	590
19.3.3	类的实现	591
19.3.4	访问对象	592
19.3.5	this 指针	592
19.3.6	new 和 delete	593
19.3.7	静态成员	594
19.3.8	只读成员	594
19.3.9	复制构造函数	595
19.3.10	友元	596
19.4	类的继承	597
19.4.1	继承的语法	597
19.4.2	继承方式	597



19.4.3	多重继承	598
19.5	函数和操作符重载	598
19.5.1	函数重载	599
19.5.2	操作符重载	601
19.6	覆盖与虚函数	604
19.6.1	覆盖	604
19.6.2	虚函数和多态	604
19.6.3	虚函数的实现	606
19.6.4	纯虚函数与抽象类	606
19.7	名字空间	607
19.8	模板	608
19.8.1	模板函数	608
19.8.2	模板类	609
19.9	异常处理	609
19.10	C 与 C++ 混合编程	611
第 20 章	嵌入式 GUI 编程	612
20.1	建立开发环境	612
20.2	简单的 Qt 应用程序	614
20.2.1	编写源代码	615
20.2.2	编译	615
20.2.3	工程文件	616
20.2.4	运行	617
20.2.5	移植到目标机	618
20.3	窗口布局	619
20.3.1	水平布局与垂直布局	619
20.3.2	栅格布局	622
20.4	Qt 对象	623
20.4.1	层次化管理	624
20.4.2	信号与槽	625
20.4.3	事件	628
20.4.4	定时器	631
20.5	使用 designer	633
20.5.1	窗体设计	633
20.5.2	代码编写	635
20.5.3	运行结果	639
20.6	Qt 常用类	640
20.6.1	QChar	640
20.6.2	QString	641
20.6.3	QPoint	647
20.6.4	QSize	648
20.6.5	QRect	649
20.6.6	QFont	650
20.6.7	QPixmap	651
20.6.8	QIcon	652



20.6.9	QWidget	654
20.6.10	QDialog	662
20.6.11	QLabel	664
20.6.12	QAbstractButton	665
20.6.13	QPushButton	667
20.6.14	QCheckBox	667
20.6.15	QRadioButton	668
20.6.16	QLineEdit	669
20.7	Qt 综合应用	670
20.7.1	软件设计	670
20.7.2	源码实现	673
20.7.3	运行结果	680
20.7.4	Qt 国际化编程	680
第 21 章	嵌入式数据库编程	684
21.1	基本 SQL 语句	684
21.1.1	数据库与表	684
21.1.2	创建和删除表	685
21.1.3	插入、修改及删除记录	685
21.1.4	条件表达式	686
21.1.5	数据库查询	687
21.2	建立 SQLite3 开发平台	688
21.3	SQLite3 编程接口	689
21.3.1	打开和关闭数据库	689
21.3.2	执行 SQL 语句	690
21.3.3	查询数据库	691
21.4	使用 SQLite3 工具	691
21.5	SQLite3 数据库应用实例	692
21.5.1	使用 sqlite3_exec 查询数据库	693
21.5.2	使用 sqlite3_get_table 查询数据库	694
第 22 章	产品开发实例：无线信息终端	696
22.1	总体架构	696
22.2	硬件设计	697
22.3	软件设计	698
22.3.1	总体框架	698
22.3.2	协议报文格式	699
22.4	应用软件	701
22.4.1	GUI 应用模块	701
22.4.2	通信协议模块	705
22.4.3	业务功能模块	710
22.4.4	使用多线程读取设备	713
22.4.5	模块集成	714
附录 A	缩略语	716



Part

第 1 部分 嵌入式开发基础

第 1 章 引言

第 2 章 C 语言编程要点

第 3 章 开发环境



第 1 章 引言

嵌入式系统 (Embedded System) 可以简单地定义为“以专门应用的实现为中心、以计算机技术和网络通信技术为基础, 软件可按需增添和裁剪, 硬件可按需配置, 系统可靠性、成本、体积、功耗要求严格的专用计算机系统”。

1.1 嵌入式产业概况

“嵌入式系统”一词最早是指用于工业和交通设备内部的控制装置, 例如数控机床中的工业控制器 (工控机)、飞行器的自主导航装置, 是一种完成专门功能的特殊计算机, 其核心功能是控制, 同时也具有一定的计算功能。随着上世纪 90 年代末以来嵌入式处理器、嵌入式操作系统和嵌入式应用开发环境及开发工具的飞速发展, 特别是与网络通信技术、信号处理技术、多媒体技术等不同技术领域的相互融合, 嵌入式技术已经成为一种集自动控制、网络通信、信号处理、分布式计算、人机互动、智能感知等功能为一体的综合性技术, 其应用范围越来越广泛, 从传统的工业和设备控制逐步拓展到家用电器、通信设备、汽车电子、医疗卫生、个人娱乐、智能家居、楼宇控制、办公自动化、商业设施、银行终端等与人们日常工作、生活密切相关的很多领域。常见的嵌入式产品包括:

- ◆ 2G, 3G 手机
- ◆ WiFi 路由器和 AP (Access Point)
- ◆ 家用以太网路由器和交换机
- ◆ 智能电表、水表、气表
- ◆ 商业 POS 机
- ◆ 数字电视机顶盒
- ◆ 高档儿童玩具 (如遥控汽车)
- ◆ 网络下载播放器
- ◆ 手持多媒体娱乐产品 (MP3, MP4 等)
- ◆ 电子词典
- ◆ 学习机
- ◆ 高档液晶电视
- ◆ 蓝光 DVD 和传统 DVD 播放器
- ◆ 手持和车载导航仪
- ◆ 家庭机器人 (如自动扫地机)
- ◆ 数码相机
- ◆ 汽车娱乐装置
- ◆ 汽车控制装置 (如 ABS, EBD 的控制系统)
- ◆ 高档智能家电
- ◆ 家庭安防系统
- ◆ 银行 ATM 机
- ◆ 银行自助缴费终端
- ◆ 城市信息机
- ◆ 智能卡系统
- ◆ 中高端打印机
- ◆ 商用复印机
- ◆ 手持电脑 (如 PDA、Mini 上网本等)
- ◆ 医疗设备 (如 B 超、多普勒彩超等)
- ◆ 测试仪表 (如数字存储示波器、逻辑分析仪、频谱分析仪、网络分析仪、协议分析仪等)
- ◆ 工业机器人、机械手



如表 1.1 所示是智能手机与台式计算机的一个简单对比。

表 1.1 智能手机与台式计算机的简单对比

比较的特性	智能手机	台式计算机
CPU	固化在板卡上，不可更换	可升级、更换
内部存储	固化在板卡上，不可更换、扩充	内存、硬盘均可升级、更换和扩充
外部存储	最多 1 个 Mini USB 接口和 1 个 SD 卡接口	多组 USB 接口、PCI 总线，提供更大的扩展余地
通信接口	2G、3G、蓝牙、WiFi	以太网、串口（及并口），通过 USB 接口或 PCI 总线相应接口板卡或模组可以扩展实现 2G、3G、蓝牙、WiFi、ADSL、红外等
人机互动	实时性要求高	实时性差，系统和程序启动需要等待很常见
维护	日常免维护	需要定期清理集尘
体积	手持便携，严格要求	基本无要求
功耗	电池操作，严格要求	基本无要求
可靠性	严格要求，死机蓝屏很罕见	基本无要求，死机蓝屏常见

从表中可以看出嵌入式系统和传统计算机系统主要有如下区别。

- ◆ 系统资源严格受限。
- ◆ 体积、功耗严格受限。
- ◆ 功能实时性要求高。
- ◆ 具有高可靠性且免维护。

应该说这些区别来自于嵌入式产品的各种应用要求，也构成了嵌入式系统的基本特性。除此之外，嵌入式产品还有如下三个非常重要的特性。

- ◆ 具体应用及其应用环境纷繁复杂。
- ◆ 各种产品的具体功能千差万别。
- ◆ 外设、总线和接口种类极其丰富。

正是这些因素，决定了相对于传统计算机系统的产品开发和应用开发，嵌入式系统的软硬件产品开发和应用开发更为复杂多变，技术要求更高、更全面。

在国内，嵌入式系统中常见的操作系统主要有以下几种：

- | | |
|------------|--|
| ◆ Linux | ◆ Palm OS |
| ◆ Symbian | ◆ Windows CE/Windows Mobile |
| ◆ uC/OS II | ◆ Windows XP Embedded/Windows Embedded |
| ◆ VxWorks | |
| ◆ Nucleus | |

在上述嵌入式操作系统中，Linux 是用途最广、装备产品最多的，特别是最近两三年间，从智

能手机和数字机顶盒到很多行业终端和工业控制设备,应用越来越普及。相比之下,其他操作系统的应用多集中在某个单一市场和单一类型的设备上。**Symbian** 仅用于手机产品,而且只有诺基亚、索尼爱立信、三星等 **Symbian** 手机联盟企业的产品才使用(并且不是全部产品都使用)。**uC/OS II** 主要用于某些实时性要求较高的小型终端设备上。**VxWorks** 主要用于某些实时性和可靠性要求很严格的大型装备或设备的控制系统和部分通信设备板卡上。**Nucleus** 主要用在那些采用中国台湾地区联发科(**MTK**)公司芯片的手机上,基本上都是中低端的非智能机。**Palm OS** 在国内很少用到,部分进口 **PDA** 产品有使用的。**Windows CE** 主要用于机顶盒等用户操作界面不太复杂的装置。**Windows Mobile** 主要用于智能手机和 **PDA** 产品,日本、中国台湾和大陆地区厂家的部分智能手机均采用该系统。**Windows XP Embedded** 是微软公司新近推出的嵌入式操作系统,主要用于银行 **ATM** 机之类的所谓立式终端设备,目前尚未见到规模应用。

Android 是什么?

Android 不是一种新的操作系统,它是以 **Google** 公司为主的开放手机联盟(**Open Handset Alliance**)提供的一个免费和开放的手机平台,主要包括基于 **Linux** 的智能手机操作系统和基于 **Java** 的应用开发环境(**SDK**, **Software Development Kit**)。

在国内,嵌入式系统常用的 **CPU** 体系有:

- | | |
|------------------|-------------------------|
| ◆ ARM | ◆ ColdFire (68K) |
| ◆ PowerPC | ◆ Intel 8051 |
| ◆ MIPS | ◆ Atmel AVR |

在上述 **CPU** 体系中,**ARM** 内核系列芯片是目前用途最广、产品出货量最大的。据不完全统计,配置有 **ARM** 内核的各种芯片最近几年的年出货量均在上亿片,特别是在 2008 年全球金融危机爆发的大背景下,基于 **ARM** 内核的芯片全球出货量取得了超过 95% 的增长,达到了 2.6 亿片。在手机市场中已牢固占据超过 80% 的市场,在家庭路由器、数字电视机顶盒、手持多媒体娱乐产品、电子词典和学习机、手持和车载导航装置、汽车娱乐装置、智能家电、银行自助缴费终端、智能卡系统、**PDA**、小型医疗设备和小型测试仪表市场中也将逐渐占据主导地位。特别是随着 **ARM** 平台上网本的出现,传统的笔记本电脑市场将很可能出现轻薄便携笔记本电脑被 **ARM** 产品替代的趋势。相比之下,**PowerPC** 和 **MIPS** 主要用于网络协议处理和某些工业控制场合,供货厂家数量远不及 **ARM** 芯片供货厂家数量,特别是因为它不支持 **Windows CE** 等操作系统,所以产品应用非常有限。**ColdFire**(即原来摩托罗拉公司半导体部的 68000 系列)用途更少、用量更小,其知识产权拥有者及核心供应商飞思卡尔公司在 **CPU** 市场的主打产品是 **ARM** 系列芯片和 **PowerPC** 系列芯片。**Intel 8051** 和各种 51 系列芯片以及 **Atmel AVR** 芯片都属于功能相对简单的单片机产品,虽然用途也十分广泛、芯片出货量也很大,但其产品和应用开发难度低、技术含量低、软件附加值低,而且基本上不支持操作系统,主要应对那些智能化要求比较低、控制功能单一的应用,产品的性价比无法与 **ARM** 系列芯片相提并论,不但不可能与 **ARM** 系列芯片进行竞争,而且现在和未来很长一段时间内还面临 **ARM Cortex-M** 系列芯片的有力挑战。

从产业发展的角度看，嵌入式产业最近几年发生了一系列重大变化，集中表现在以下方面。

- ◆ 越来越多的半导体厂家涉足嵌入式芯片，从 CPU 到嵌入式系统专用的 Mobile DDR，包括 Intel（英特尔）、AMD、NXP（飞利浦半导体）、TI、三星电子、现代电子、VIA 等全球半导体行业中位居前列的诸多巨头。
- ◆ 越来越多的软件厂家开始发力嵌入式软件开发，不但大量的中小公司从事嵌入式 Linux 操作系统和驱动的开发，微软这样的行业巨无霸也连续推出多款嵌入式操作系统产品——Windows CE/Windows Mobile，Windows XP Embedded，Windows Embedded for Point of Service，Windows Vista for Embedded Systems。
- ◆ 越来越多的传统家电厂家和计算机厂家将其产品重心逐步转移到嵌入式产品系列上来，其典型代表就是依靠 iPhone 和 iPod 赚得盆满钵盈的美国苹果公司，而国内各大计算机卖场和家电卖场里也充斥着手持娱乐、电子导航、电子词典、学习机、PDA 等五花八门的嵌入式产品。
- ◆ 越来越多的传统通信设备厂家开始进军嵌入式产品及其开发领域，例如 2008 年诺基亚全资控股的奇趣科技（Trolltech）一举就显示出其在嵌入式软件开发领域欲与微软一争高下的雄心。
- ◆ Google 以开源方式推出基于嵌入式 Linux 系统的 Android 嵌入式操作系统，吸引了大批品牌和山寨手机、上网本厂家进入嵌入式 Linux 产品领域。

综观全局，嵌入式产业正在进入持续快速发展的产业成长期。

关于一个产业发展趋势的基本判别准则

- ◆ 这个产业所提供的产品和服务与人们日常的工作和生活是否越来越密切相关？出现在我们身边的概率是不是越来越大？
- ◆ 这个产业里是不是大公司不少，中小公司也很多？有没有出现很多夕阳产业那种小公司基本绝迹、大公司越来越少的现象？

1.2 Linux 操作系统

Linux 操作系统对很多读者来说已经不再陌生了，这是一种已经得到广泛应用的计算机操作系统。它是由计算机业界大名鼎鼎的芬兰人林纳斯·托瓦兹（Linus Torvalds）最早开发的。从 1991 年的 Linux 0.01 版本到 1994 年的 Linux 1.0 版本发展至今，本书交稿时内核版本是 Linux 2.6.30。

在实际中，人们提到“Linux”一词时其实并不是单纯地指操作系统，而是泛指以下三个部分的总体概念。

- ◆ 一种类 UNIX、名为“Linux”的计算机操作系统。
- ◆ Linux 操作系统环境下的开发工具和开发环境。



◆ Linux 操作系统环境下的各种应用软件和工具软件。

在详细介绍 Linux 操作系统之前，有必要简单介绍一下目前已经与 Linux 密不可分的 GNU 计划（GNU Project），首先强调两点。

◆ Linux 内核开发不是 GNU 计划的一部分。

◆ Linux 诞生之时（1991 年），GNU 计划已运作多年，大部分基本工具软件已开发出来。

GNU 是理查德·斯托曼（Richard Stallman）在 1984 年创立的，其目标是发展一个完全免费的自由软件——一个 UNIX 类计算机操作系统以及运行在其上的软件开发工具和各种应用程序。具体实施是由自由软件基金会（Free Software Foundation, FSF）来负责的。但在实际发展中，GNU 自己的操作系统内核并没有真正开发出来并得到应用，反倒是大量 GNU 软件（包括开发工具、调试工具和各种各样的应用软件等）与 Linux 操作系统及其开发和应用完美地融合在一起了——GNU 软件运行在 Linux 内核之上，整个 Linux 内核基于 GNU 通用公共许可，甚至与各种各样的 Linux 版本一同发行和使用。

Linux 发展至今，其应用已十分广泛。在传统计算机领域，大型计算机（以及超级计算机）、工作站和服务器市场的主流操作系统是 Linux，有消息称，2008 年 11 月全球最快的超级计算机前 500 名中，使用 Linux 操作系统的有 439 个，即 Linux 操作系统在当今超级计算机市场的占有率接近 90%，这是计算机发展史上前所未有的。而在台式机用户中，使用 Linux 操作系统的也越来越多，特别是国内外很多计算机高手和黑客都以使用 Linux 为骄傲。在笔记本电脑这一 Windows 传统市场中，也有越来越多公司开始在自己的产品上预装 Linux 系统。但是，当今的传统计算机行业却不是 Linux 最大的市场——若以装备产品的种类和数量以及市场销售额而言，嵌入式系统无疑是 Linux 现在和今后最大的市场。目前几乎所有种类的嵌入式产品都有使用 Linux 的，例如 Linux 在智能手机领域中已经与 Symbian, Windows Mobile（Windows CE）形成三足鼎立之势，在数字机顶盒、家庭路由器、行业应用终端等诸多领域也已稳固占据主流地位。

目前有很多公司提供各种各样的 Linux 发行版，主要是台式机桌面系统版本，部分也提供服务器系统版本，Debian 5.0 还提供 ARM 平台版本。其中绝大多数公司同时提供 32 位版本和 64 位版本。国内常见的 Linux 发行版主要有：

◆ Debian

◆ Ubuntu

◆ Fedora

◆ Red Hat

◆ Slackware

◆ Mandriva

◆ SUSE

◆ Knoppix

◆ CentOS

◆ TeamLastOS

大多数发行版一般都会包含以下内容：

◆ Linux 内核

◆ GNU 函数库和开发工具软件

◆ 图形化应用开发环境（Qt）

◆ 图形界面环境（如 KDE 或 GNOME 用

◆ 基本应用软件（如 OpenOffice，Firefox 等）

◆ 多国语言包及其输入法

◆ 程序源代码



户应用环境)

除了这些 Linux 发行版外,很多公司原本用于 Windows 系统的很多商业软件也开始提供 Linux 版本,例如数据库软件中的 Oracle 和 DB2、EDA 软件中的 Mentor Graphics PADS 和 Cadence Allegro、科学计算软件中的 MathWorks Matlab 和 Intel Fortran Compiler 等。另外,根据 GNU 授权规则,任何人都可以采取收费或免费的方式来发行 Linux 操作系统下的软件,并在符合该授权的规范下修改其他 GNU 软件,因此就吸引了大量程序开发人员把原本运行在其他操作系统下的软件免费程序移植到了 Linux 上。特别是互联网的深入普及极大地方便了 Linux 软件程序的获取和发布,使得 Linux 下各种各样的大小软件数量种类极为庞大、功能五花八门。另外,随着 KDE 和 GNOME 图形环境的日臻完善, Linux 操作系统在用户图形环境方面已经与 Windows 难分伯仲。即使在嵌入式领域中,Qt 图形开发套件的不断成熟也使得越来越多的需要图形化操作界面的嵌入式产品有了新的选择。

从技术开发的角度看, Linux 软件开发与 Windows 软件开发有很大的区别。本书的大部分读者,可能更多地关心嵌入式 Linux 软件开发。嵌入式 Linux 软件开发所面临的主要技术挑战性是由 Linux 操作系统自身的一些特点和嵌入式系统的各种具体产品应用所决定的。从软件开发者的角度看, Linux 操作系统的基本特点可以简单地总结为以下三条。

- ◆ 硬件体系支持广泛。
- ◆ 软件源代码开放。
- ◆ 内核版本不完全考虑前向兼容。

就支持的硬件体系而言,迄今为止还没有一种操作系统能够与 Linux 相比,以 Linux 2.6.30 版本的内核为例,其所支持的 CPU 体系包括:

- | | |
|------------------------------------|----------------------------|
| ◆ Alpha AXC (DEC 公司,全球首款 64 位 CPU) | ◆ M68K (即 ColdFire,飞思卡尔公司) |
| ◆ AMD x86-64 | ◆ MIPS |
| ◆ ARM | ◆ PA-RISC (惠普公司) |
| ◆ AVR32 (Atmel 公司) | ◆ PowerPC |
| ◆ BLACKFIN (ADI 公司) | ◆ IBM S/390 |
| ◆ H8/300 (瑞萨科技) | ◆ SUN SPARC |
| ◆ Intel IA64 | ◆ x86 |
| ◆ M32R (瑞萨科技) | ◆ Xtensa (Tensilica 公司) |
| | ◆ SuperH (日立公司) |

上面提到这些 CPU 体系中大多数实际上并不为绝大多数开发人员所熟知,如果再看一看 Linux 内核文件系统中 driver 子目录的内容,可以说 Linux 对各种接口、外设和总线等林林总总硬件的支持程度早已远远超出大多数技术人员的想象。显然,如此完整的硬件支撑体系,一方面给设计开发人员提供了巨大的选择余地和操作空间,使得“广阔天地,大有作为”成为可能,另一方面也构成了一种技术挑战——如何才能够根据客户的需要和自己的实际情况正确地选择一个合理的、能够持续开发应用的基本硬件平台架构?这一点也是市场上同一类嵌入式产品各家公司实现方案迥异的根本原因之一。若单纯考虑软件方面的因素,开放源代码的确给开发人员提供了极其丰富

的代码资源,为其代码编写和程序开发提供了相当多的帮助和便利,而如何结合客户要求及自身开发能力与习惯、根据所选硬件体系结构,在浩瀚的代码中选择出适合自己产品开发和应用的软件功能模块并加以剪裁、组合、移植及优化却是一项既复杂而又需要仔细做的工作。当然 Linux 内核版本的更新导致不兼容旧版本这一令多数开发人员颇为不爽的惯例,还使得旧版本的开发经验有时候不能在新版本的开发中提供直接有效的帮助,例如 Linux 2.6.26 内核中移去了应用极其广泛的触摸屏的驱动支持,就使得原来的相关开发工作和经验很难得到借鉴。

根据本书几位作者的个人经历与感受,不得不说嵌入式 Linux 软件开发非常富有技术挑战性,乐趣无穷而又充满痛苦。其实这种感受在嵌入式开发人员中颇为普遍,看一看网络上很多开发人员写的博客就会体会到,但是有时候开发中的郁闷、成功后的喜悦溢于言表。

当然,相对于其他嵌入式操作系统下的各种开发,特别是 Windows 平台上的嵌入式开发(主要是 Windows CE/Windows Mobile 开发),嵌入式 Linux 在技术和应用开发方面的优势还是非常明显的。以 Windows 平台上嵌入式开发为例,其存在的以下问题严重地制约了开发人员的工作开展。

- ◆ Windows Mobile 支持的 CPU 平台很少,目前支持 x86、部分 ARM 内核版本、部分 MIPS 内核版本、部分 SuperH SH4 版本。Windows XP Embedded 则只支持 x86。
- ◆ 嵌入式系统中应用最多的是 ARM 体系,目前 Windows Mobile 只支持 ARM 的 v4, v4T, Thumb, v5TE 和 Xscale,对 ARM 高版本的支持还有很多欠缺。
- ◆ 基本开发工作严重依赖微软开发平台(Visual Studio .NET 和 Windows CE/Windows Mobile PlatformBuilder)和芯片厂家的板级支持包(Board Supporting Package)——缺乏其中任何一个的支持,即使很简单的功能都无法实现。
- ◆ 由于源代码并不开放,所谓的软件优化基本无从谈起。

1.3 ARM 体系概况

ARM (Advanced RISC Machine) 是一家专注于通用处理器知识产权 (Intellectual Property, IP) 研究开发的公司,自身并不直接生产处理器芯片,而是通过向其他芯片和半导体制造商提供设计授权来推广其产品,其产品和 CPU 技术体系架构在业内被通称为 ARM 处理器。

ARM 处理器在技术体系上属于更为纯正的精简指令体系(RISC),在具体实现架构上与 x86 体系完全不同——ARM 处理器多数采用的是 SoC (System on Chip, 片上系统) 架构,CPU、图形/图像/视频处理单元、显示单元、存储控制器及其接口、基本通信单元(UART, SPI, I2C, I2S 等)的控制器以及许多外部总线或接口控制器(如以太网 MAC 控制器、USB 控制器等)都集成在单一芯片上。因此采用 ARM 处理器的计算机没有一般 x86 体系上独立的南、北桥芯片和图形加速卡、以太网卡控制器芯片等,整体架构更为精简,更适合于嵌入式应用场合。

ARM 处理器体系功能集除了核心的 ARM 功能集外,还有如下常见的功能集。

- ◆ Thumb: 采用 16 位指令集,其中的大部分指令被直接映射成正常 ARM 指令,这样使得 CPU 能够通过运行 16 位指令来实现很多需要 32 位指令才能完成的功能,从而提高代码密度,节省处理器和内存资源。
- ◆ Thumb-2: Thumb 功能集的扩展,通过增加的 32 位指令来扩展指令执行位宽。

- ◆ Jazelle: 通过在 ARM 结构中直接执行 Java 字节码来提高 Java 程序的运行性能。
- ◆ ThumbEE (Thumb 执行环境): 也称为 Jazelle RCT (Runtime Compilation Target), 是对 Thumb-2 功能集的修订和扩展, 实现了更好的实时编译功能——包括即时编译 (Just in time compilation, JIT) 和动态自适应编译 (Dynamic Adaptive Compilation, DAC)。
- ◆ VFP (Vector Floating Point): ARM 架构上的矢量浮点处理器扩展, 实现满足 ANSI/IEEE Std 754-1985 标准的单精度/双精度浮点运算能力。
- ◆ NEON (Advanced SIMD): 64 位/128 位混合 SIMD (Single Instruction Multiple Data, 单指令多数据) 高级扩展功能, 支持 8 位、16 位、32 位和 64 位整数和单精度浮点数据运算, 以及完整的 SMP (对称多处理) 功能, 可同时进行 16 个运算操作。
- ◆ TrustZone: 安全扩展功能。
- ◆ DSP 指令扩展: 对原有算术处理指令进行了进一步扩展, 并增加了单时钟周期 16×16 和 32×16 乘法/加法器指令。

ARM 处理器按其官方分类包括应用处理器 (Application processor)、嵌入式处理器 (Embedded processor)、安全内核 (SecurCore) 和图形处理器 (Graphics processor) 以及视频引擎 (Video engine)。

应用处理器主要针对开放的操作系统平台上的业务应用更为广泛和丰富的市场需求, 目前包括以下内核版本系列:

- | | |
|-----------------|--------------------|
| ◆ ARM720T | ◆ ARM1176JZ(F)-S |
| ◆ ARM920T | ◆ ARM11 多核 |
| ◆ ARM922T | ◆ ARM Cortex-A8 |
| ◆ ARM926EJ-S | ◆ ARM Cortex-A9 单核 |
| ◆ ARM1136J(F)-S | ◆ ARM Cortex-A9 多核 |

嵌入式处理器主要针对嵌入式体系中那些实时性要求较高、实现功能较为确定单一的市场需求, 目前主要包括以下版本系列:

- | | |
|--------------|--------------------|
| ◆ ARM7EJ-S | ◆ ARM1156T2(F)-S |
| ◆ ARM7TDMI | ◆ ARM Cortex-M0 |
| ◆ ARM7TDMI-S | ◆ ARM Cortex-M1 |
| ◆ ARM946E-S | ◆ ARM Cortex-M3 |
| ◆ ARM966E-S | ◆ ARM Cortex-R4(F) |
| ◆ ARM968E-S | |

安全内核主要包括以下系列:

- ◆ SC100
- ◆ SC200
- ◆ SC300

主要图形处理器目前只有 Mali 处理器一个系列, 视频引擎也只有 Mali-VE 一个系列。

ARM 公司对其处理器内核版本传统的命名方式是“内核版本号+扩展功能集标识”，内核版本号为 3 或 4 位数字，扩展功能集标识如表 1.2 所示。

表 1.2 扩展功能集标识

标识	说明
T	支持 Thumb 功能集（T2 表示支持 Thumb2）
D	支持片上调试（Debug）功能
M	支持片上乘法器
I	支持嵌入式在线仿真和嵌入式跟踪调试
E	具备 DSP 增强功能
J	支持 Jazelle 功能
F	支持 VFP 功能
-S	支持多种扩展功能的整合版本

在嵌入式行业中，一般将 ARM 整个体系的处理器都视为嵌入式处理器，大致分为嵌入式应用处理器（Embedded Application Processor）和微控制器（Microcontroller 或 MCU）。很多获得 ARM 公司授权的公司，例如 Freescale（飞思卡尔）公司、Atmel 公司等，对自己的 ARM 处理器产品也并不严格遵守 ARM 公司的分类方法，多数按照 ARM 处理器的 IP 版本划分，分成 ARM7 产品、ARM9 产品、ARM11 产品、Cortex 产品等。下面将分别介绍不同 IP 版本 ARM 处理器的基本情况。

1.3.1 ARM7

ARM7 系列采用低功耗 32 位 RISC 处理器内核，具备 MMU（Memory Management Unit，内存管理单元）和高速片上缓存（cache）的内核版本能够支持 Windows CE，Palm OS，Symbian OS，Linux 以及部分实时性操作系统，其指令前向兼容 ARM9，ARM9E 和 ARM10E（以及基于 ARM10E 的 Intel Xscale 架构）。目前主要包括以下 4 个版本：

- ◆ ARM720T
- ◆ ARM7EJ-S
- ◆ ARM7TDMI
- ◆ ARM7TDMI-S

获得 ARM7 系列内核授权的半导体厂家非常多，芯片用途非常广泛，用量也相当大。常用的 ARM7 芯片有以下一些。

- ◆ NXP 公司：LH75xx 系列（ARM7TDMI-S）、LH79 系列（ARM720T）、LPC21xx 系列（ARM7TDMI-S）、LPC22xx 系列（ARM7TDMI-S）、LPC23xx 系列（ARM7TDMI-S）、LPC24xx 系列（ARM7TDMI-S）、LPC28xx 系列（ARM7TDMI）。
- ◆ Atmel 公司：AT91SAM7xxx 系列、AT91SAM42800 A/55800A、AT91x40 系列（ARM7TDMI）。
- ◆ 三星电子：S3C44B0，S3C3410（ARM7TDM）。
- ◆ 德州仪器（TI）：TMS470（ARM7TDMI）。
- ◆ 意法半导体公司：STR71x/73x 系列（ARM7TDM）和 75x 系列（ARM7TDMI-S）。

由于功能相对较少、处理能力相对较弱，ARM7 系列的 CPU 在实际中多用于简单控制，常见于低端手机（入门级非智能手机）、入门级的 MP3/MP4、低端工业控制器等产品，其产业角色更接近于高档 32 位单片机。按照 ARM 公司的技术路线，ARM7 系列未来将逐渐被 Cortex-M 系列所替代。

1.3.2 ARM9

目前包括 ARM920T 和 ARM922T 两个内核版本，采用 32 位 RISC 处理器内核和 5 级整数流水线，并集成指令和数据缓存以及一个 32 位 AMBA 接口，全部具有 MMU，能够支持 Symbian OS, Palm OS, Linux 和 Windows CE。

获得 ARM9 系列内核授权的半导体厂家不是很多，基本上都是 ARM920T 内核，常用芯片有如下一些。

- ◆ 三星电子：S3C241x 系列、S3C244x 系列。
- ◆ Cirrus Logic 公司：EP93xx 系列。
- ◆ Atmel 公司：AT91RM9200。

该系列产品虽然芯片种类相对较少，但芯片产量很大、用途非常广泛，商业和银行 POS 终端、智能家电以及嵌入式教育培训用板卡等都会用到该系列内核的芯片。

这里需要特别指出一点，Cirrus Logic 公司的 EP93xx 系列产品实际上并不是传统意义上的 ARM 芯片，而是该公司在 ARM920T 内核基础上进行了很大改进和增强的一款复合功能芯片，在硬件上实现了数学协处理器、2D 图形加速器和加密器等实际应用中非常必要的功能，因此其整体功能明显优于传统意义上的 ARM9 和 ARM9E 内核的芯片。

1.3.3 ARM9E

ARM9E 采用 5 级整数流水线架构，并提供 VFP9 功能增强选项，同时还使用了 TCM（Tightly-Coupled Memory）技术；其数据和指令的缓存容量可变（0MB~1MB）且具有独立的 AMBA AHB 总线接口；特别是在原有 ARM9 支持的 ARM 和 Thumb 功能（指令）集基础上增加了 DSP 支持，还通过单时钟周期 32×16 乘法/加法器进一步强化了 16 位定点性能，属于具备 DSP 功能的 32 位 RISC 处理器。目前主要包括以下版本：

- | | |
|--------------|-------------|
| ◆ ARM926EJ-S | ◆ ARM968E-S |
| ◆ ARM946E-S | ◆ ARM996HS |
| ◆ ARM966E-S | |

ARM9E 系列内核的产品应用极其广泛，如中高端非智能手机、低端智能手机、工业控制产品、汽车电子设备等。获得 ARM9E 系列内核授权的半导体厂家很多，绝大多数都是 ARM926EJ-S 内核授权，常用的 ARM9E 芯片有如下一些。

- ◆ 三星电子：S3C2450。
- ◆ Atmel 公司：AT91SAM9 系列。

- ◆ TI 公司: Da Vinci 系列 (Da Vinci TMS320DM 系列) 和 OMAP-L137。
- ◆ NXP 公司: LPC31xx 系列和 LPC32xx 系列。
- ◆ Freescale (飞思卡尔) 公司: MCIMX21 和 MCIMX27。
- ◆ Cirrus Logic 公司: EP9608。

1.3.4 ARM10E

ARM10E 仅有 ARM1020E 一个内核版本, 它采用 6 级流水线架构, 内置 MMU、数据和指令缓存, 以及 64KB 片上一级缓存 (32KB 数据 + 32KB 指令), 并集成了一个支持嵌入式在线电路仿真实时逻辑 (Embedded ICE-RT) 的整数处理单元, 配置了外部协处理接口和系统内部协处理器 CP14 和 CP15, 整数处理单元及其缓存、写缓冲、系统总线接口和协处理器之间采用了全 64 位接口, 还通过独立的指令侧和数据侧 64 位 AHB (Advanced High performance Bus, 高性能总线) 接口实现了多级结构 AHB 支持, 另外还改进了电源管理功能。

获得 ARM10E 内核授权的半导体厂家很少, 市场上一般只能见到 Marvell 公司的 PXA27x 系列芯片, 它采用的是在 ARM1020E 内核基础上进行了很多特殊扩展的 Intel Xscale 体系, 中国台湾地区 and 大陆地区很多手机厂家的 2G 智能手机都采用这个体系的芯片, 还有不少公司用其开发手持电脑、学习机等产品。

1.3.5 ARM11

ARM11 系列是 ARM 公司推出的高端产品。所有产品都采用 ARM v6 版本的处理器内核、64 位片上缓存接口, 集成了一个支持 Embedded ICE-RT 逻辑的整数处理单元, 配置了系统内部协处理器 CP14 和 CP15, 片上高速 AMBA (Advanced Microcontroller Bus Architecture, 高级微控制器总线结构) 二级接口支持多处理器实现, 还可选集成一个数学协处理器支持 VFP。针对不同的市场定位包含以下 4 大类。

- ◆ ARM1136JZ(F)-S: 针对视频流解码进行了扩展, 8 级流水线架构, 并提供外部协处理器接口, 主要用于视频解码要求较高的场合。目前产品有飞思卡尔公司的 MCIMX31/35/37 系列和德州仪器公司的 OMAP24xx 系列, 主要用于中档 3G 手机和高档汽车娱乐设备。
- ◆ ARM1156T2(F)-S: 9 级流水线架构, 具备指令与内存保护单元 (Memory Protection Units, MPUs), 并提供外部协处理器接口, 可选集成的一个 VFP 数学协处理器专门解决精密控制中的高精度数学运算问题。主要用于汽车控制系统和实时性要求很高的工业控制系统。
- ◆ ARM1176JZ(F)-S: 支持 TrustZone 安全扩展和智能电源管理, 8 级流水线架构, 可配置的低级中断潜伏 (Latency), 数学协处理器专门应对 3D 图形加速功能实现了多媒体处理功能的扩展, 主要用于多媒体功能要求较高的场合 (如高档智能手机灯)。目前主要产品有三星电子的 S3C64xx 系列 (美国苹果公司的 iPhone 手机采用的就是该系列处理器)。
- ◆ ARM11 MPCore: ARM11 的多核产品, 通过 AXI (高级 AMBA) 二级接口连接多个单核 CPU, 最多支持 4 个 ARM11 CPU (每个 CPU 都配备了私有定时器和看门狗), 配置了一个缓存轮询控制单元来实现不同 ARM11 CPU 片上数据缓存间的数据协同, 同时具有分布式中断控制功能支持不同 ARM11 CPU 的中断潜伏, 还支持所有的 RAM 事例奇偶校验。

1.3.6 Cortex

Cortex 系列采用 ARM v7 版本的处理器内核，是 ARM 公司新近推出的智能处理器（Intelligent Processors）系列，也是其未来一段时间的主推产品。目前已经有不少公司推出了相应产品。该系列主要包括三类。

- ◆ Cortex-A 用于完成复杂业务功能（如 3G 智能手机、高清视频处理灯），支持 Linux，Windows Mobile 等功能完整的嵌入式操作系统。其中 Cortex-A8 和 Cortex-A9 内核支持 ARM，Thumb 和 Thumb-2 指令集，目前主要包括以下版本。
 - Cortex-A8：产品有德州仪器公司的 OMAP3xxx 系列、飞思卡尔公司的 MCIMX51，目前 2G 手机市场国际主流品牌的很多智能手机都采用 OMAP3xxx 系列芯片。
 - Cortex-A9 MPCore（多核）：产品有德州仪器公司的 OMAP4xxx 系列（单芯片集成 2 个 Cortex-A9 MPCore），目前 3G 手机市场国际主流品牌的高档智能手机都采用该系列芯片。
 - Cortex-A9 SingleCore（单核）
- ◆ Cortex-R 用于实时性要求更高的系统，支持 ARM，Thumb 和 Thumb-2 指令集，目前主要包括以下版本。
 - Cortex-R4：产品有德州仪器公司的 TMS570M。
 - Cortex-R4F
- ◆ Cortex-M 是低成本、低功耗应用的内核，仅支持 Thumb-2 指令集，目前主要包括以下版本。
 - Cortex-M0
 - Cortex-M1
 - Cortex-M3：迄今为止，取得 Cortex-M3 内核授权的芯片厂商已超过 28 个，2008 年全球半导体厂家排名前 10 位的厂家中有 6 个获得授权，已经量产供货的芯片超过 250 款，典型产品有德州仪器公司的 LM3Sxxxx 系列、意法半导体公司的 STM32 系列、NXP 公司的 LPC13xx 和 LPC17xx 系列、Atmel 公司的 SAM3U 系列。

1.4 常见嵌入式产品及其基本平台简介

1.4.1 学习开发板

我们日常生活和工作中经常接触的嵌入式产品其实很多，但学习开发板可能是很多读者意识中的第一种嵌入式产品。顾名思义，学习开发板是用来学习如何进行嵌入式开发的板子，而业内所说的开发板或评估板则主要用于产品和系统开发中软硬件平台的选择评估、软件开发和硬件设计。一般来说，学习开发板应该为用户提供不同典型版本的操作系统和开发环境，例如 Linux 2.6.26 版本和 2.6.30 版本的内核，Qt 2.3 和 Qtopia 1.7，Qt Embedded 4.5.x 和 Qt Extended（Qtopia）4.4.x，以及各个软件版本详细的开发操作手册和源代码，方便用户学习演练。开发板或评估板则

应为用户提供除硬件技术资料以外诸如技术支持、工业级设计参考、增值功能设计等服务。

根据笔者了解到的情况,学习开发板的客户大致可以分为两类:嵌入式技术爱好者和嵌入式开发技术新手。

嵌入式技术爱好者大都是在校学生,也有部分对嵌入式技术有兴趣爱好的在职人员,他们的特点是没有产品开发意识,个人兴趣爱好占主导地位,没有特定的技术目标和具体的技术要求,时间较为充裕,经常上网交流,喜欢把弄或尝试各种各样版本的内核、驱动、应用开发环境等。嵌入式开发技术新手大都是刚刚参加工作或者原来从事 IT 其他领域工作的在职人员,他们的特点是工作或者职业发展上很可能对嵌入式产品开发知识和技能的掌握有比较明确的要求,这种要求可能来自公司的岗位要求,也有可能是自己的职业调整需要,对学习开发板的技术指标和功能等有比较具体明确的期望或要求(例如期望通过学习开发板做出来的东西或者学到的东西能够迅速应用到自己的日常工作之中)。严格地说,这两类客户应该选择不同的学习开发板。对于嵌入式技术爱好者,市场上常见的基于三星 S3C2410 和 S3C2440 平台的学习开发板是比较合适的,如图 1.1 所示是北京诚捷鸿远通信技术有限公司提供的学习开发板 HY2410A,它采用 S3C2410A 芯片作为处理器。对于嵌入式开发技术新手,可能更多地应该考虑针对某一行业、与自己日常工作需要或职业发展需要相关的产品开发平台。本章后面几节将重点介绍几种针对特定行业应用的嵌入式产品开发平台。

三星 S3C2410 和 S3C2440 平台的共同优点是:硬件系统成熟稳定、价格低廉, Linux 操作系统和 Windows CE (以及 Windows Mobile 5.0) 系统的内核与驱动支持比较完备,嵌入式应用中常见的总线 and 接口多数都已配置,如 UART (Universal Asynchronous Receiver/Transmitter, 通用异步接收/发送接口)、USB Host 和 USB Device、液晶屏、触

摸屏、立体声音频、以太网、GPIO (General-Purpose IO, 通用可编程输入/输出)、SPI (Serial Peripheral Interface, 串行外设接口)、I2C (Inter-Integrated Circuit)、I2S (Inter-IC Sound)、SD/MMC (SD, Secure Digital Memory Card; MMC, Multi-Media Card), 以及 AD (analog-to-digital, 模数变换)、定时器、实时时钟 (Real Time Clock, RTC)、DMA (Direct Memory Access, 直接内存存取) 控制器、看门狗等,特别是因为芯片进入市场较早,所以使用者、开发者众多,互联网上公开的硬件设计资料和软件移植经验非常多,初学者获得各种帮助更加容易。当然,这两种平台也有共同的缺点,例如串口 (UART) 不是全功能的,USB 只支持 1.1 版本,而且官方开发资料获取不易。

S3C2410 和 S3C2440 平台的核心架构和主要配置是完全一样的, S3C2440 除了主频更高、显示和 SD/MMC 功能有所增强、GPIO 端口数有所增加外,相比于 S3C2410 增加的只有以下功能。

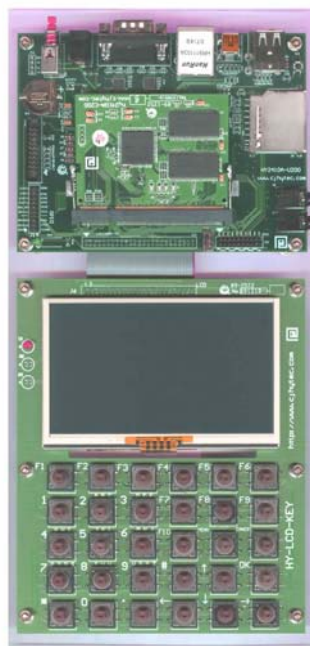


图 1.1 HY2410A 开发板外观

- ◆ 支持 2K Byte 页面的 NAND Flash。
- ◆ 提供摄像头接口。
- ◆ 支持 AC'97。
- ◆ 提供用于触摸屏接口的内部 FFT (Fast Fourier Transform, 快速傅里叶变换)。

由于两个平台都使用 ARM920T 内核,且无任何特别扩展,因此总体性能的实际差距并不大。例如, S3C2440 虽然配置了摄像头接口,但其内核没有像 ARM9E 系列那样内置 DSP 功能,外部也没有数学协处理器、图形加速器或 DSP 核扩展,因此虽然工作主频是 S3C2410 的两倍,但在图形图像方面的处理性能提升却非常有限——其实计算机发展史早已证明,在 CPU 内核架构不变的前提下单纯依靠提高 CPU 工作频率来获得性能的显著提高是不可能的。

现在市场上的 S3C2410 和 S3C2440 学习开发板种类型号非常多,选择时应注意以下问题。

- ◆ 厂家提供的资料 and 软件是否齐全有效,对于软件源代码要特别小心,部分厂家把其他公司的软件不加编译通过、板级运行和测试就冒充自己的产品提供给用户。
- ◆ 尽可能选择同时配置了液晶屏和键盘的产品,液晶屏最好自带触摸功能,键盘最好类似于手机键盘,方便自己学习使用。
- ◆ 尽可能要求现场检查厂家提供的 Linux 内核和驱动程序、Qt (Qtopia) 等软件能不能正常编译通过,能不能在板子上正常运行,主要外设接口是否工作正常。
- ◆ 厂家有没有能力提供多个 Linux 内核版本和应用软件开发环境,如支持简体中文的 Qt Extended (Qtopia) 4.4.x 和 Qt Embedded 4.5.x,以及无线上网 (2G, 3G, WiFi) 等常见应用组件。

由于 S3C2410 和 S3C244x 系列推出较早,ARM 内核版本较旧,加上三星公司在设计上的一些考虑有待商榷 (例如没有集成以太网 MAC、USB 仅支持 1.1 标准等),比较适合一般性的嵌入式技术学习、教学和实验。如果真正想进行与产品开发有关的工作或者希望更深入地探求嵌入式开发技术,基于 ARM926 内核芯片 (如 Atmel 公司的 AT91SAM9263、飞思卡尔公司的 MCIMX25x、三星公司的 S3C2450) 的开发板是比较好的选择,有更高要求的可以考虑 TI 公司出品的集成 ARM 和 DSP 的双核平台系列 (OMAP-L137, OMAP-L138, OMAP-3530) 或者 ARM11 内核系列芯片 (如飞思卡尔公司的 MCIMX31 和 MCIMX35x,三星公司的 S3C6410)。

建 议

嵌入式技术学习:入门选择三星的 S3C2410,有较高追求的考虑三星的 S3C2450 或者 S3C6410,对 ARM 和 DSP 都有追求的选择 TI 公司的 OMAP-L137。

商业应用和工业控制:选择 Cirrus EP93xx 和 Atmel AT91SAM926x,高端应用选择飞思卡尔的 i.MX37。

消费电子:入门级选择三星的 S3C2450,中低端选择 TI 公司的 OMAP-L137,高端选择 TI 公司的 TMS320DM365、飞思卡尔的 i.MX31 和三星的 S3C6410。

汽车电子:低端选择飞思卡尔的 i.MX25,高端选择飞思卡尔的 i.MX35。

1.4.2 行业终端

市场上常见的嵌入式行业终端有很多种，日常生活中接触最多的是连锁超市、便利店、专卖实体店及百货商场、餐饮网点（如美食城）、音像图书门店等各种零售企业使用的商业 POS（Point of Sale）机（或商业 POS 收款机）和各个银行网点中越来越多的银行自助终端。

商业 POS 机一般包括具备以下功能的硬件实体。

- ◆ 双显示屏：一个显示屏供操作员使用，另一个显示屏为大尺寸屏幕，向顾客显示其所购买的物品信息及价格信息。部分行业使用的 POS 机还要求在无交易进行时关闭操作员使用的显示屏，同时通过大尺寸显示屏播放广告。
- ◆ 网口：用于连接 POS 机和服务器（或数据中心）。
- ◆ 多个串口：用于连接条码扫描枪、磁条卡读卡器、IC 卡读卡器和密码键盘。
- ◆ 打印机：一般为打印交易凭条（小票）的热敏打印机，发票打印机很少见。
- ◆ 操作键盘：一般至少有 16 键。
- ◆ 密码键盘：主要用于银行卡的刷卡密码信息保护。部分行业使用的 POS 机还要求有内置硬件加密装置以保护顾客的商品交易信息。

银行自助终端一般包括具备以下功能的硬件实体。

- ◆ 带触摸功能和防爆功能的大屏幕：用于显示交易操作信息，无交易时可播放广告。
- ◆ 网口：用于连接终端和银行主机服务器。
- ◆ 内置磁条卡读卡器、IC 卡读/写卡器和密码键盘。
- ◆ 打印操作记录凭证的窄行打印机。
- ◆ 监控摄像头。
- ◆ 现钞出纳机。
- ◆ 部分终端内置硬件加密装置：用以保护除交易密码以外的用户其他操作信息的通信。

传统商业 POS 机和银行自助终端大都采用单片机方案或者 x86 体系的工控机方案（多采用 x86 体系中的 i386, i486, Pentium 等古老 CPU），不但体积大、功耗高、可靠性差、维护不易，而且不支持嵌入式操作系统，软件更新困难。最近几年越来越多的商业 POS 机厂家和银行终端厂家采用 ARM 处理器来装备其产品，体积、功耗大大降低，软件更新升级更为便捷，运营和维护支持大大减少。特别是很多不涉及现钞交易的银行终端（例如主要提供自助缴费、自助转账、自助查询功能的终端），已经可以完全做成壁挂型产品甚至薄板型产品，其作业网点已经从银行街面点逐步进入居民小区和工矿企业厂区等银行卡用户密集地域，极大地拓展了银行营运网络的实际覆盖范围。

Cirrus Logic 公司出品的 EP93xx 系列芯片中，EP9307 和 EP9315 是两款非常适合行业终端产品开发的 ARM 处理器，其主要特性及应用优势如下。

- ◆ 内置数学协处理器，整数和浮点数标准运算以及常用信号处理指令直接由硬件实现，极

大地减轻了 CPU 内核的运算压力,显著提升了芯片的总体处理能力,特别适合于需要加解密和普通音视频处理的场合,如银行自助终端中图片或音视频广告的播放。

- ◆ 内置 2D 图形加速器,显著增强了 2D 图形处理和显示功能。再加上支持液晶显示和标准 VGA 双显示输出,非常适用于双显示和大屏显示应用需要,如商业 POS 机和银行自助终端。
- ◆ 内置硬件实现加密单元,每个芯片拥有全球唯一的 32 位 ID,方便信息加密的快速实现和终端操作的原始记录与跟踪。
- ◆ 3 路 UART(通用异步收发传输器)支持多个串口设备的通信,如条码扫描枪、磁条卡/IC 卡读卡器和密码键盘与商业 POS 机之间的通信。
- ◆ 3 路全速 USB 2.0 Host 端口支持更多外设接入以及更多扩展,例如大容量固态 SSD(Solid-state device)的使用。
- ◆ 多键键盘和触摸屏的支持,结合大屏幕显示,方便了用户操作。
- ◆ 工业级芯片支持零下 40°C 到零上 85°C 的正常工作温度区域。

由于数学协处理器和 2D 图形加速器都是在 ARM 体系之外采用专门硬件来实现的,因此 EP9307 和 EP9315 的实际性能要比 S3C2410 和 S3C2440 等单纯采用 ARM920T 内核的芯片强大很多,甚至与很多采用 ARM926EJ-S 内核的芯片相比较也有一定的优势。这里需要说明的是,EP9307 和 EP9315 芯片所内置的数学协处理器、2D 图形加速器和加密单元均与 ARM 公司无关,是 Cirrus Logic 公司自行开发的。EP9307 和 EP9315 的主要特性对比如表 1.3 所示。

表 1.3 EP9307 和 EP9315 主要特性对比

		EP9307	EP9315
核 心 架 构	内核	ARM920T	ARM920T
	主频	200MHz	200MHz
	数学协处理器	√	√
	2D 图形加速器	√	√
	硬件加密器	√	√
外 部 接 口	1/10/100Mbit/s 自适应以太网	1 路	1 路
	UART 端口	3 路	3 路
	USB 2.0 全速 Host 端口,支持 OHCI	2 路	2 路
	IDE	×	1 路,支持 2 个器件
	PCMCIA	×	1 路
	红外	1 路	1 路
	液晶显示输出	1 路	1 路
	VGA 显示输出	1 路	1 路
	触摸屏	4/5/7/8 线模拟电阻型	4/5/7/8 线模拟电阻型
	键盘	最多支持 64 键	最多支持 64 键
	I2S 接口	1 路	1 路



	AC'97 接口	1 路	1 路
	SPI 接口	1 路	1 路



由于 Cirrus Logic 公司 EP93xx 系列芯片的架构较为特殊，原厂软件版本也比较老（Linux 2.6.8，Qt Embedded 2.3），在其上进行 Linux 2.6.26 以后版本的内核移植开发和 Qt Embedded 4.5/Qtopia Extended 4.4 的嵌入式开发有较大技术难度。

1.4.3 工业控制

工业控制是集控制技术、测试测量技术、计算机技术、网络通信技术等多种技术为一体的综合技术应用领域，主要用于生产过程和生产设备的检测、控制、调度和管理，实现安全生产、连续稳定运行以及降低原材料消耗和能源消耗等目的。工业控制系统的工作环境非常复杂，应用场合很可能存在强电磁干扰、化学腐蚀、高温高湿或低温干燥等恶劣条件，而且不同行业的生产过程和设备对工业控制系统的具体功能要求千差万别，各种现场总线和接口类型繁多，因此工业控制系统的要求非常高，主要表现在以下方面。

- ◆ 恶劣环境下的长期可靠性和稳定性高。
- ◆ 分布式控制，并具备更好的检错容错机制。
- ◆ 实时性好，且抗外界变化能力强。
- ◆ 功耗低，易于在设备内部进行供电。
- ◆ 体积小巧，易于装配在设备内部。
- ◆ 接口适配能力以及相应的软件驱动能力强。
- ◆ 具备远端编程调试和维护能力，减少维护调试造成的停产。

传统的工业控制中大都采用基于 PC 机的软硬件体系，主要是 x86 体系的 PC 机，一般功耗较高、体积较大，维护更新周期短、停产时间长，特别是很多应用场合不得不采用集中控制的模式，极大地增加了生产的控制风险，已经越来越不适应现代生产对工业控制的要求。最近几年，工业控制中越来越多地采用基于嵌入式体系和分布式控制的技术，要求控制系统具备远程更新、诊断和维护功能，尽可能减少对生产流程和设备运行造成中断。目前，基于 ARM 内核 CPU 体系和嵌入式 Linux 系统的工业控制装置正在得到日益广泛的应用。

Atmel 公司出品的 AT91SAM9263 是一款非常适合工业控制应用的嵌入式 CPU 平台，其关键特性如下。

- ◆ 技术成熟，平台可靠性和稳定性好。从 8 位、16 位到 32 位，Atmel 公司的微控制器产品在工业控制领域已经得到长期的大量应用，其产品工艺和质量久经考验。
- ◆ 对实时性功能的硬件支持好。例如主频 200MHz 时就可以达到 220MIPS 的性能指标。
- ◆ 低电压供电设计和片上功率管理，降低了平台总体功耗。
- ◆ 硬件平台体积小巧，其最小系统的体积已经能够做到普通 U 盘容量大小。
- ◆ 配置有 CAN 现场总线接口和多组 32 位可编程并行接口，接口适配能力突出。
- ◆ 除提供 JTAG、ETM(Embedded Trace Macrocell)和 2-线 UART 调试接口并支持 RT-ICE 功能外，所提供的 USB 接口、UART 接口和以太网口均可下载软件，极大地方便了远程



软件调试和维护。

该平台的其他主要特性如下。

- ◆ ARM926EJ-S 内核结合硬件实现的 2D 图形加速器，总体性能强大。
- ◆ 接口类型十分丰富，除支持 Part 2.0A 并兼容 Part 2.0B 的 CAN(Controller Area Network, 控制器局域网) 接口外，还提供：
 - 2 个 USB 全速 Host 接口和 1 个 USB 全速 Device 接口。
 - 摄像头接口（支持国际电信联盟 ITU-R BT. 601/656）。
 - 液晶显示接口和 6 通道 AC'97 音频接口。
 - 3 个 UART、1 路 I2C、1 路 I2S。
 - 2 个 SPI 接口、1 个 TWI (Two-wire Interface, 双线) 接口。

Atmel 公司为 AT91SAM9263 配套的资料和软件都可以从其官方网站上免费下载，这就大大方便了一般性的开发工作，开发成果还可以很快应用到 AT91SAM9260，AT91SAM9261 和 AT91SAM9262 等芯片上。

1.4.4 手持娱乐

首先声明，本书所说的手持娱乐产品并不包括 PSP 之类的游戏机。

一般意义上的手持娱乐主要是指音乐播放、视频播放，很可能也包括一些电子小游戏。按照这一概念，手持娱乐涵盖的产品范围就太广了，甚至是不是能够成为单独一类功能界定清晰的产品都可能是一个问题了。传统意义上的手持娱乐产品主要指 MP3/MP4 播放器，而现在很多手机、PDA、GPS 导航、MID (Mobile Internet Device)、收音机、电子词典、数码相框、学习机、电子阅读器、读书机等都具备音频、视频文件播放能力，甚至还有能播放 MP3 的小型空气净化器，应该说手持娱乐已经成为很多小型大众消费电子产品的一项基本功能。

手持娱乐产品的基本要求总体上可以概括为以下三条。

- ◆ 需要支持的媒体文件格式和编解码方式多，而且可能越来越多。
- ◆ 体积小、重量轻、功耗低。
- ◆ 外部存储接口支持完善。

当然，支持较大尺寸（例如 3.5/4.3/5 英寸）和较高分辨率（例如 800 像素×600 像素）的彩色液晶显示屏也是必需的。

以往很多具备手持娱乐功能的产品都采用硬件解码器的方案，其优点是解码速度快、输出质量高，但缺点也很明显——支持的多媒体文件格式和编解码方式是固定的，更新和扩展比较困难。而现在多媒体技术的一大特点就是编码方式多、文件格式也多，而且越来越多。甚至某些文件格式中具体的编解码参数也有不小差异，造成同一类型（即通常所说的文件名后缀相同）的文件不是都能够用同一种播放器（或播放软件）播放。因此，越来越多的产品采用软件解码的技术结构，其优势之一就是可以通过升级软件来升级所支持的多媒体文件格式和编解码方式。

采用软件解码的产品大体上有两种技术方案可选：一种是选择高性能嵌入式处理器，例如选择

ARM1176JZF-S 或 ARM Cortex-A8/A9 平台；另一种是选择异架构双核平台，即其中一个核是传统的 DSP，另一个核是传统的 ARM。针对大多数手持娱乐产品中音视频文件播放为主要应用这一特点，后一种方案更为可取，前一种方案更适合 MID、智能手机那些要求综合功能更多的手持设备。

TI 公司的 OMAP 系列嵌入式应用处理器中，OMAP-L137 是一款比较适合中低档手持娱乐产品的开发平台，而基于 TI 达芬奇技术的 TMS320DM365 则比较适合于对视频播放、即时导航以及视频与图像拍摄等功能有更多需求的高端手持娱乐产品的开发。

OMAP-L137 平台的主要特色如下。

- ◆ 采用异架构双核体系，一个核为带有 DSP 增强功能的 ARM926EJ-S 内核，另一个核为 TI 的 C674x DSP 核，两个内核的工作主频均为 300MHz。显然，强大的 DSP 功能保证了该平台能够提供更好的多媒体播放功能，而且通过软件更新还可以不断地升级或更新播放软件。
- ◆ BGA 管脚数较少、管脚（植球）间距较大，PCB 板面积小，设计制造成本低。
- ◆ 功耗低，芯片满负荷工作情况下的功耗不足 500 毫瓦。
- ◆ 支持 SD/MMC，特别是提供包含 PHY 的 USB 2.0 OTG，极大地方便了外部文件的获取——既可以在 Host 模式下从 U 盘、USB 硬盘盒等提取文件，也可以在 Device 模式下从计算机向设备下载文件。
- ◆ 配置了 SDIO (Secure Data I/O) 接口，方便与 WiFi、蓝牙等通信模块的互联，更好地支持不同设备/计算机之间的无线互联和文件共享。
- ◆ 外设接口集成度很高，片上集成了以下接口：
 - 3 个 UART 接口 (16500 型，其中 1 个带 Modem 控制)。
 - LCD 显示输出。
 - 2 个 SPI 接口。
 - 2 个 I2C 接口。
 - 1 个 HPI (Host-Port Interface) 接口。
 - 1 个 USB Host (OHCI) 1.1 接口 (包含 PHY)。
 - 3 路多媒体音频输出。
 - 10/100Mbit/s 以太网 MAC 接口。

TMS320DM365 平台的主要特色如下。

- ◆ 多核体系，包括一个 300MHz 主频的 ARM 926EJ-S 内核和 2 套基于达芬奇技术的视频/图像处理引擎 (硬件支持 H.264, MPEG4, MPEG2, MJPEG, JPEG, WMV9/VC1 编解码)。
- ◆ 视频子系统硬件支持实时图像处理。
- ◆ 支持图像传感器接口 (Image Sensor Interface, 简称 ISIF) 和 CMOS 摄像头接口 (兼容 BT.601/BT.656/BT.1120、数字化 YCbCr 标准)。
- ◆ 自带透镜失真校准功能。
- ◆ 硬件支持直接写屏显示。
- ◆ 提供 NTSC/PAL 电视标准编码输出。

- ◆ 支持 8/16 位 YCC 和 24 位 RGB888 数字显示输出。
- ◆ 内置 LCD 显示控制器。
- ◆ 10/100Mbit/s 自适应以太网 MAC。
- ◆ 2 路 SD/MMC 接口, 并支持 SDIO 接口。
- ◆ 支持 DDR2, mDDR, NAND Flash, NOR Flash, OneNAND, SmartMedia/xD 卡。
- ◆ 片上还集成了以下接口:
 - 2 个 UART 接口。
 - 1 个 I2C 接口。
 - 5 个 SPI 接口。
 - 1 个 USB 2.0 Host 高速接口和 1 个 USB 2.0 OTG 高速接口。
 - 语言编码接口。

由于这两款芯片都是 ARM+DSP 的双核架构, 所以开发工作需要配套 TI 公司的专业开发软件才能顺利进行, 而且如何优化 ARM 和 DSP 的双核性能也是比较有挑战性的工作。

1.4.5 医疗仪器

在医疗仪器中, 嵌入式系统除了实现一般的控制功能外, 非常重要的功能是医学图像处理, 而且随着临床诊断和病理分析的需要愈来愈强烈, 医学影像的信息处理技术在医疗领域的作用也愈来愈显著。

常见的医疗影像包括常规 B 超、彩超 (彩色多普勒超声图像)、数字 X 光图像、核磁共振 (Nuclear Magnetic Resonance Imaging, 简称 NMR) 或磁共振 (Magnetic Resonance Imaging, 简称 MRI)、CT (Computed Tomography, 计算机断层扫描) 图像、各种电子和光学内窥镜 (如胸腔镜、腹腔镜) 图像、显微切片图像、PET (Positron Emission Tomography, 正电子发射计算机断层成像)、SPECT (Single Photon Emission Computed Tomography, 单光子发射型计算机断层, 简称 SPECT 或 SPET) 图像等。医疗影像的常规处理包括: 基本成像和虚拟成像、任意比例的放大和缩小、任意角度的旋转、任意位置的平移、对比度增强或边缘锐化、图像还原、三维造影、图像分割和局部提取、噪声基底滤波、图像配准、病灶点识别和特征提取等。

由于人体组织的生理和生物学特性极其复杂, 而各种医疗影像的获取手段又有着不同的缺陷, 医疗影像一般都比较模糊, 局部特征不均匀, 病理组织边界往往由于相互浸润或掩盖而变得很不清晰。这些给影像处理带来了很大难度, 因此医疗仪器中多采用具有较强 DSP 功能的嵌入式图像处理板来完成医疗影像的处理。

TI 公司的 OMAP3530 是该公司专门为医疗仪器设计开发的一款高档嵌入式图像处理和平台, TI 公司还委托第三方为其专门开发了医疗应用开发套件。该平台的主要特色如下。

- ◆ 采用异架构多核体系, 提供强大的图像、图形和视频加速功能, 包括 4 个硬件实现的处理功能单元:
 - 一个 600MHz 主频的 ARM Cortex-A8 内核的 NEON SIMD CPU。
 - 一个 430MHz 主频的、基于 TMS320C64x⁺ DSP 实现的高性能图像音视频加速处理器。
 - 一个支持 OpenGL ES 1.1/2.0 和 OpenVG 1.0 工业标准的 2D/3D 图形加速器。

- 一个高级超长指令 TMS320C64x⁺ DSP 单元。
- ◆ 3 路高速 SD/MMC 接口, 并支持 SDIO 接口。
- ◆ 2 路摄像头输入(串行、并行各 1 路), 支持 CCD 和 CMOS 摄像头以及 BT.601/BT.656 标准(YCbCr 4:2:2、8/16 位)接口。
- ◆ 双输出显示处理单元, 包括 1 路图形输出和 2 路视频输出, 支持 4 倍/8 倍显示放大和 90°, 180°, 270° 显示旋转。
- ◆ 智能电源管理, 支持动态电压/频率调谐(DVFS, Dynamic Voltage and Frequency Scaling)。
- ◆ 片上还集成了以下接口:
 - 3 个 UART 接口。
 - 3 个高速 I2C 接口。
 - 3 个 USB 2.0 Host 高速接口和 1 个 USB 2.0 OTG 高速接口。
 - 5 路多通道缓冲型串行接口。
 - 4 路主从型多通道串行接口。
 - 1 路单线(One-Wire)通信接口。

与 OMAP-L137 和 TMS320DM365 一样, OMAP3530 的开发工作也需要配套 TI 公司的专业开发软件才能顺利进行。

1.4.6 汽车电子

汽车电子是嵌入式系统最大的市场, 汽车是单一实体中装备嵌入式系统最多的。从电子装备的角度看, 汽车有如下特殊要求。

一、传感器多

汽车中需要实时检测的信息非常多, 例如发动机转速、油箱油量、水箱水温、防冻液存量、清洗液存量、车内外温度、门窗(天窗)开启、灯光开启、机油存量(机油压力)、蓄电池电压、车速、制动液存量、制动未松动、冷却液温度、胎压、制动气压、制动灯保险丝、空气流量、进气压力、节气门位置、发动机爆燃、车身位移、方向转角、尾气排放、车外环境光照强度等。

二、独立控制的模式

出于安全的考虑, 汽车不使用全车集中控制模式(包括多主从热备份的集中控制模式), 以避免核心控制系统的局部或个别故障影响整车工作和性能。在绝大多数汽车上, 进行独立控制的功能子系统有点火系统、(自动)变速箱、燃油控制、发动机(怠速)控制、防抱死制动(ABS)、防滑转(ASR)、尾气再循环、动力转向、悬架控制、防盗告警、车内空调、安全装置、门窗控制、灯光控制、车内音响。部分高档汽车还有电子导航、前视红外/激光/超声波防撞雷达、倒车雷达和倒车影像、右侧盲区影像、并线预警、座椅防撞击控制、前向防撞预警、巡航控制、四轮驱动控制、翻滚预防等。而且大部分与启动、制动、行驶、安全、尾气排放有关的功能控制是自动独立进行的, 驾乘人员不能干预其工作过程, 甚至不能对其进行开启或关闭。

三、分布式无主从通信模式

由于汽车上很多传感器和功能控制模组之间存在信息依赖关系, 因此相互之间的信息交互非常

重要。各个传感器的信息通过车载通信网络实现点到点交互,各个独立控制模块也通过车载通信网络实现点到点通信,各个通信节点(传感器、控制模组等)没有严格的主从之分、具有同等地位。目前,汽车上应用最为广泛的车载通信网络是 CAN (Controller Area Network, 控制器局域网),一般由两条 CAN 总线构成,一条高速 CAN 总线连接主要功能控制模组和组合仪表(中控板),一条低速 CAN 总线连接门窗控制、车灯和舱内照明等。

四、电磁干扰大

汽车上的发电机、启动机、电子点火系统在使用时都会产生大量电磁脉冲干扰。

五、环境及机械振动

汽车上的环境温度、湿度变化大,机械振动冲击十分频繁。

基于汽车电子的这些特别要求,没有一种微处理器或者微控制器能够承担所有的功能角色,多数汽车控制单元采用各种 8 位、16 位、32 位单片机和微处理器(主要是 ARM 体系中的 ARM7, ARM9 和 Cortex-M3 系列内核,以及 PowerPC 体系中的 e200 系列等)。最近几年,随着汽车车载娱乐、电子导航需求的迅速增长,越来越多的中高级轿车采用具备多媒体功能的中控台系统。飞思卡尔公司的 i.MX25 系列中的 i.MX255 是为中高端汽车的多媒体中控系统开发的一款 ARM 芯片,非常适合开发针对国内 20~50 万价位的 B 级车和 C 级车的车载中控娱乐系统,该芯片的主要特性如下。

- ◆ 满足国际汽车电子设备委员会 AEC-Q100 三级(Grade 3)标准(汽车应力测试标准)。
- ◆ $-40^{\circ}\text{C} \sim +85^{\circ}\text{C}$ 工作温度。
- ◆ ARM926EJ-S 内核,主频最高达 400MHz。
- ◆ 内置蓝牙协议栈,支持语音导航和车内免提电话。
- ◆ 支持 DDR2 和 mDDR、SDRAM 内存以及 NOR Flash 和 NAND Flash。
- ◆ 配置 CMOS/CCD 摄像头接口,方便倒车影像获取。
- ◆ 支持 16 位彩色 LCD 显示,最高分辨率可达 SVGA (800 像素×600 像素),支持高分辨率电子地图和触摸导航。
- ◆ 2 路音频输出——SSI/I2S 和 ESAI(Enhanced Serial Audio Interface),硬件支持 5.1 声道音响和 AAC/MP3/WMA 文件格式。
- ◆ 配置 MMC/SD/SDIO 接口、10/100Mbit/s 以太网 MAC 和 5 个 UART。
- ◆ 高速 USB OTG 接口(片上集成 PHY)和高速 USB 2.0 Host 接口(片上集成 PHY)。
- ◆ 2 套 CAN 总线控制器(FlexCAN)。
- ◆ 3 路可配置串行外设接口(C-SPI)和 3 路 I2C 接口。
- ◆ 支持 Windows Embedded CE 6.0, Linux (以及汽车级专用 Linux), QNX。

由于汽车电子应用中对功耗和体积的要求比手持设备的要求宽松很多,因此可以采用主频更高、结构更复杂的内核平台,开发中更多地应注重系统稳定性和可靠性以及电磁兼容能力,针对驾驶安全预警、车载无线通信、车载多媒体娱乐等的各种应用开发将是软件开发的重点。

1.4.7 智能本

从 2009 年初开始,面对飞速成长的上网本市场,一向表现出对传统 PC 机市场兴趣不大的

ARM 公司终于禁不住巨大的市场诱惑,开始发力笔记本电脑市场。智能本(SmartBook)就是 ARM 公司和高通(Qualcomm)公司新近推出的一种智能移动平台概念,意在从 Intel, AMD 和 VIA(台湾威盛)公司把持的传统笔记本电脑市场中夺取相当份额。当然对于智能本,不同的公司从不同的角度赋予了它不同的内涵,有的认为智能本实际上是另一种形态的上网本(也就是成本更低、屏幕更小、功能更简单的笔记本电脑),有的认为是一种集成更多 PC 功能的智能通信终端(也就是屏幕更大、更适合于便携而非手持的智能手机),我们更倾向于后一种定义。从实际的市场需求和用户习惯看,相对于传统的笔记本电脑以及所谓的上网本,智能本的核心功能更多地集中在通信和娱乐这两个方面。说简单点,智能本的基本功能就是上网、聊天、通话、看片儿,当然也应该具有一定的文档和图形图像处理以及电子游戏功能,例如编辑 Word 文档、进行 PPT 演示、浏览图片、进行视频拍照、玩 2D 小游戏等,其主要特点如下。

- ◆ 小巧轻便,超长待机(待机时间 12 小时以上)。
- ◆ 使用固态硬盘而非传统磁性存储硬盘。
- ◆ 配置 USB OTG、多合一读卡器和摄像头。
- ◆ 硬件支持多种格式的高清视频文件播放,最好具有 HDMI 接口。
- ◆ 支持以太网、WiFi、蓝牙、WiMAX、2G 和 3G 等多种无线通信。
- ◆ 使用无须散热的真正低功耗硬件平台(最好不需要热管散热)。
- ◆ 使用 Linux 操作系统,并具备图形化用户环境。
- ◆ 硬件成本应与中高档智能手机相匹敌。

飞思卡尔公司新近推出的 i.MX515 芯片是目前智能本市场的主流平台,它具备以下功能。

- ◆ 采用主频高达 1GHz 的 ARM Cortex-A8 内核,并配置了 NEON SIMD 媒体硬件加速器和 VFP 处理器。
- ◆ 硬件支持 OpenGL ES 2.0 和 OpenVG 1.1 图形图像加速。
- ◆ 硬件支持多种流媒体文件格式的 720p 高清视频解码。
- ◆ 硬件支持 DVD-1 级别的视频编码。
- ◆ 支持高达 24 位真彩色的宽屏显示(1280 像素×800 像素),还支持 18 位色彩的辅助显示。
- ◆ 720p 高清电视分量输出。
- ◆ 支持 DDR2 内存和大容量 NAND Flash。
- ◆ 高级电源管理,支持动态电压和主频调整、动态温度补偿。
- ◆ 多种外设接口,包括 10/100Mbit/s 以太网、高速 USB OTG、SDIO、SPI、I2C、UART、PATA、红外、键盘等。
- ◆ 支持 I2S 和 S/PDIF 音频接口。
- ◆ 硬件实现的安全技术/加密器,更好地支持版权保护。
- ◆ 无须风扇或热管散热,真正实现 Fanless 设计。

台湾和硕科技(Unihan Corporation,台湾华硕下属子公司)等公司已经推出了基于 i.MX515 芯片的智能本,使用的是 Linux 操作系统各个发行版本中比较流行的 Ubuntu 9.04(Jaunty Jackalope)。随着 ARM 平台智能本的市场日益扩展,将会有越来越多的 ARM 芯片厂家涉足这个

产品领域。

虽然 ARM 表面上刻意将智能本与传统的基于 x86 体系的上网本区别开来，但实际上两者的市场用户人群基本是相同的，相互之间必然出现直接而激烈的市场竞争。随着高通、飞思卡尔、德州仪器等公司 ARM 智能本芯片的推出和市场成功，ARM 与 x86 的硬件平台之争、Linux 与 Windows 的软件平台之争将愈演愈烈，IT 界又一场龙虎斗即将上演。

1.5 嵌入式产品开发基本流程

中小型公司的嵌入式产品开发基本流程如图 1.2 所示。

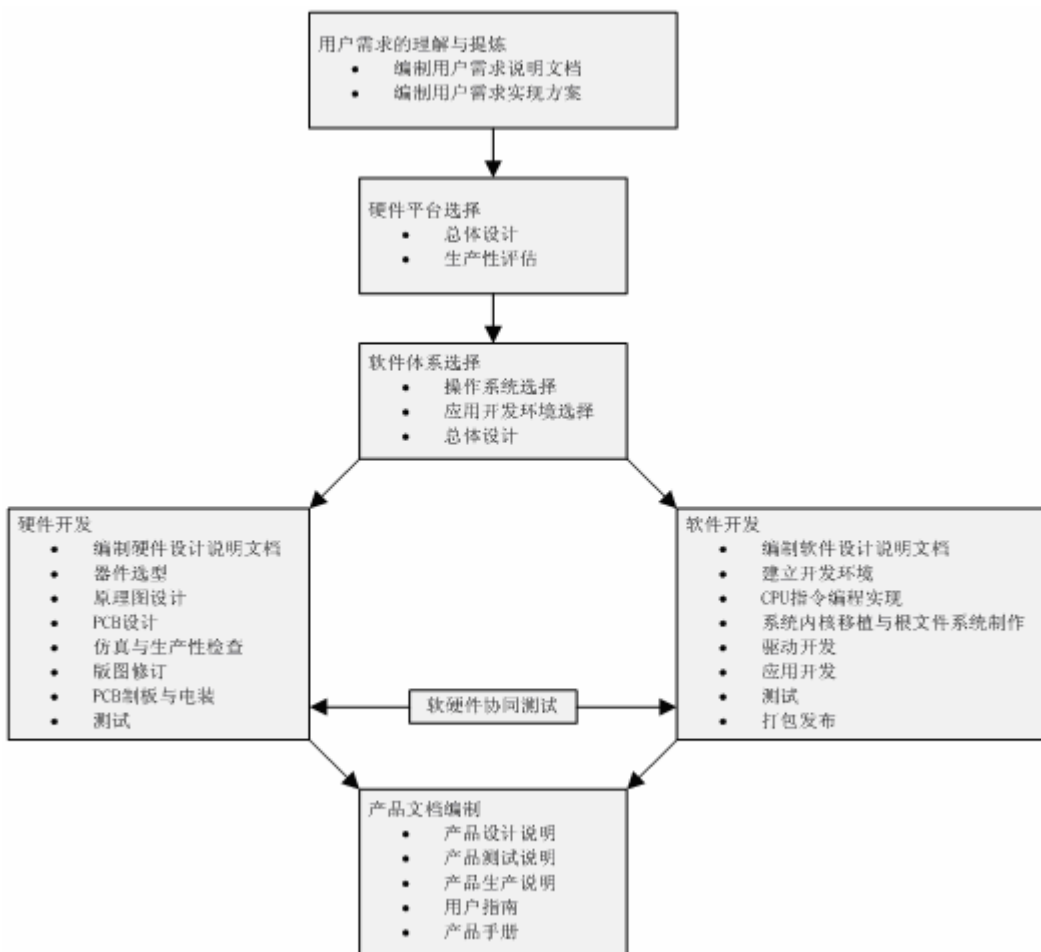


图 1.2 嵌入式产品开发流程

我们有点经验和理解可以在这里和读者们一起交流一下。

一、关于用户需求的理解和提炼

- ◆ 了解用户的行业背景、用户自己所提供的产品和服务是与用户进行有效沟通的基础。
- ◆ 理解用户需求是一个反复多次的过程。

二、关于硬件平台的选择

- ◆ 实用性是第一位的，不要盲目追求平台的新颖性，中小公司开发实际产品最好使用业内流行的、稳定成熟的平台，尽可能不要使用最新的平台——除非你另有企图（例如融资需要）。
- ◆ 可生产性非常重要，与你的供应商、PCB 制板厂家、电子装配厂家进行充分的沟通非常关键——他们的能力制约着你的能力。实际中经常遇到的很多细节问题很可能就是你面对的一道道“门槛”。

三、关于软件体系的选择

- ◆ 开发人员熟悉什么开发环境或者什么开发工具是选择的基础——要知道从不懂到会做远比从低水平提升到高水平难度大得多。
- ◆ 开发人员有途径获得有用的开发资源或者经验参考对开发工作有着巨大的促进作用——特别是免费的网络资源。
- ◆ 软件极有可能涉及到版权许可问题，例如在选择 Windows Mobile 作为产品软件平台之前一定要搞清楚：都有哪些软件包可能需要付费，愿不愿意、有没有能力缴纳那些费用，一个中小公司的软件开发工作能够得到多少来自微软的有效技术支持。
- ◆ 选择业内一般所说的开源开发环境 Qt 时，请注意 Qt 实际上包含两种体系。一种是名为 Qt 的应用开发框架，包括开发工具和运行环境。另一种是名为 Qt Extended 的应用平台和用户接口，它除开发工具和运行环境外，还集成了很多应用软件（类似于 Windows 环境下的那些应用软件，如输入法、记事本、游戏、计算器、系统管理器、媒体播放器、网络终端等）；目前该平台已被诺基亚停止开发，公开发行版本仅支持英语，其他语言包需要单独付费；目前简体中文语言包（即支持中文处理的 Qt Extended 简体中文版）已经有第三方公司能够提供（www.cjhytec.com）。两种体系的各个历史版本如表 1.4 所示。

表 1.4 Qt 体系及其历史版本

产品名称	基本功能	历史版本
Qt	C++ 应用开发框架	Qt 4.5.0 / Qt Embedded 4.5.0
		Qt 4.4 / Qt Embedded 4.4
		Qt 4.3 / Qtopia Core 4.3
		Qt 4.2 / Qtopia Core 4.2
		Qt 4.1 / Qtopia Core 4.1
		Qt 4.0
		Qt 3.3（平台和编译器）
		Qt 3.2
		Qt 3.1
		Qt 3.0
		Qt 2.3
Qt Extended	嵌入式 Linux 应用平台与用户接口	Qt Extended 4.4
		Qtopia 4.3
		Qtopia 4.2
		Qtopia 4.1
		Qtopia 2.2

Qtopia 2.1

Qtopia 2.0

Qtopia 1.7

Qtopia 1.6

四、关于文档

- ◆ 每一次成功之后，把自己成功历程中的每一步及时地用文字详细记录下来是良好的开发习惯之一，而定期的文档整理和反思，将会使你受益更多。一定要记住：人的记忆是日益模糊、逐渐淡忘的，只有清晰的文字记录才是持久和可靠的，因此不要把开发经验建立在个人记忆之上。
- ◆ 通过文档撰写，能够在逻辑思维的高度，对开发中遇到的问题做进一步的厘清，对自己的解决方案做更清晰的思考，有助于发现潜在的问题并启迪新的思路。



- ◆ 请仔细阅读厂家的各种技术文档并进行交叉参考、对比，厂家的文档是由很多技术人员共同撰写的，他们都是人，都可能犯错误。
- ◆ 定期去厂家网站更新自己的技术资料，厂家那里被其他客户玩出来的 bug 一定会比你板子上跑出来的多。



第 2 章 C 语言编程要点

C 语言作为一种系统级的编程语言，其重要性是毋庸置疑的。在 Linux 操作系统上，几乎所有的系统调用都直接提供 C 语言接口；在 Linux 内核编程领域，C 语言和汇编语言更是无法替代的首选语言。

在 Linux 系统上，一般使用 GNU 的 C 编译器作为开发工具。GNU C 的语法除了兼容国际标准 ISO C90 之外，有自己的一些扩展，这些扩展的语法经常在内核编程中用到。因此本章除了介绍标准 C 语言之外，也穿插介绍了一些 GNU C 的语法。由于篇幅有限，C 语言的各种精微之处不可能一一尽叙，本章主要根据笔者在开发中的经验和体会，着重介绍实践意义比较大的内容。

本章可供对 C 语言有一定了解的读者拾遗补阙，以便进一步理解和学习 Linux 系统编程以及内核编程。需要指出的是，C 语言有若干个不同的标准，本章中凡是提到 C 语言标准的地方，指的都是 ISO (International Organization for Standardization, 国际标准化组织) C90 标准。

C 语言中将 `/*` 和 `*/` 这对符号包围起来的内容作为注释，不过 GNU C 也允许在代码中使用双斜杠 `//` 作为注释内容的标记。在本章中，我们仍然遵循 C 语言的传统格式对代码进行注释。

2.1 数据类型

C 语言是一种“强类型”的语言，程序中的各种数据都有确定的类型。编译程序时，编译器要对数据的类型进行检查，如果与所需的类型不匹配，就会给出警告或者错误信息，这是避免程序出现错误的重要手段之一。

C 语言中的基本数据类型可分为整数和浮点数两种，分别如表 2.1 和表 2.2 所示。

表 2.1 整数类型

类型名	字节数	取值范围	备注
signed char	1	-128 ~ 127	有符号字符型
unsigned char	1	0 ~ 255	无符号字符型
signed short	2	-32 768 ~ 32 767	有符号短整型
unsigned short	2	0 ~ 65 535	无符号短整型
signed int	2 或 4	同 signed short 或 signed long	有符号整型
unsigned int	2 或 4	同 unsigned short 或 unsigned long	无符号整型
signed long	4	-2 147 483 648 ~ 2 147 483 647	有符号长整型
unsigned long	4	0 ~ 4 294 967 296	无符号长整型

所有的整数类型都包括有符号和无符号两种，其中有符号类型名称中的 `signed` 关键字可以省略，默认为有符号类型。有符号类型可以表示负数、零和正数，无符号类型只能表示零和正数。



有些编译器将名称前没有 `signed` 关键字的类型默认为无符号类型，这种情况下有符号类型的名称前必须加上 `signed` 关键字。

`int` 型数据的长度没有明确规定，但现在主流的计算机平台（包括 `x86` 和 `ARM`）上，编译器都将其处理为 4 个字节，即跟 `long` 型一样。本书中如不加特别说明，`int` 型都默认为 4 个字节。

表 2.2 浮点数类型

类型名	字节数	备注
<code>float</code>	32	浮点型
<code>double</code>	64	高精度浮点型

浮点数在计算机内部由表示有效数字的部分和表示小数点位置的部分组成，前者的位数越多则精度越高，后者的位数越多则取值范围越大。浮点数类型不分有符号和无符号，同一类型既能表示正数又能表示负数。



GNU C 支持 8 个字节的 `long long` 整数类型和 12 个字节的 `long double` 浮点数类型，它们的取值范围更大，精度更高。

2.2 常数

常数是直接写在程序代码中的数据，它的值是程序编译时就可以确定的，在程序运行时不会发生变化。常数可以分为整数、实数、字符等。在 C 语言中，它们都有各自的表示方法。

一、整数的表示

C 语言支持以十进制、十六进制、八进制的形式表示整数，例如整数 100 和 65535 用十进制分别写为如下形式：

```
100    65535
```

用十六进制则写为如下形式：

```
0x64    0xFFFF
```

前面要加上 `0x` 以与十进制形式相区别，用八进制则写为如下形式：

```
0144    0177777
```

前面要加上一个 0 以与十进制形式相区别。



C 语言中的十进制数与数学上的写法完全相同，但如果前面加了多余的 0，则会被当成八进制数处理。

按以上形式书写的常数，其类型为 `signed int` 型。如果要表示无符号型数据，可在后面加一

个字母 `u` 或 `U`，如果要表示长整型数据，则加一个字母 `l` 或 `L`，两者也可以同时使用，表示无符号长整型数据，如：

```
100u    0x64L    0144uL
```

二、实数的表示

数字中如果包含小数点，则被视为实数，如：

```
1.0    3.1415926
```

要注意即使 `1` 和 `1.0` 在数学上是相等的，在 `C` 语言里也是两个类型不同的常数。

还有一种科学计数法表示的实数，如：

```
3e8    1e-6
```

字母 `e` 后面是一个整数，表示要乘以 `10` 的多少次幂。小写字母 `e` 也可以换成大写字母 `E`。按以上两种形式书写的常数，其类型均为 `double` 型。如果要表示 `float` 型数据，可在后面加一个字母 `f` 或 `F`，如：

```
1.0f    1e-6F
```

三、字符的表示

用一对单引号把一个可见字符包围起来就表示它是一个字符常数，如：

```
'a'    'A'    '3'    ' '
```

其中最后的一个字符是空格。

有些字符不能直接放在单引号内，必须用一个反斜杠 `\` 加另外一个字符来表示，这种表示方法称为转义字符。`C` 语言中常用的转义字符如表 2.3 所示。

表 2.3 常用的转义字符

换行符	<code>\n</code>	回车符	<code>\r</code>	问号	<code>\?</code>
制表符	<code>\t</code>	走纸符	<code>\f</code>	单引号	<code>\'</code>
垂直制表符	<code>\v</code>	响铃	<code>\a</code>	双引号	<code>\"</code>
退格符	<code>\b</code>	反斜杠	<code>\\</code>		

一种更直接的方式是用反斜杠和字符的 `ASCII` 码（`American Standard Code for Information Interchange`，美国信息交换标准代码）值来表示一个字符，这里的 `ASCII` 码值可以用八进制或十六进制表示，例如：

```
'\141'    '\101'    '\x33'    '\x20'
```

其中以 `\x` 开头的表示后面是十六进制数，否则表示是八进制数。这时要求十六进制数的位数不大于 `2`，八进制数的位数不大于 `3`。

字符常数的类型都是 `char` 型。

2.3 变量

变量在形式上是代码中的标识符，它的值是在程序运行时才确定的，并且可以被修改，修改变量的值称为对变量进行赋值操作。实质上，每个变量都对应着一个或多个内存单元，用以保存它的值，故给变量赋值又称为写操作，使用变量的值又称为读操作，读写操作又统称为访问。

2.3.1 变量的定义与初始化

变量在定义时就确定了它存放的数据类型，即变量的类型。定义变量时只需给出它的类型和名称即可，如：

```
int i, j; /* 定义了两个整型变量，名称分别是 i 和 j */
unsigned char new_char; /* 定义了一个无符号字符型变量 new_char */
float f1, f2; /* 定义了两个浮点型变量 f1 和 f2 */
```

变量名称与常数的区别在于它的第一个字符必须是字母或下画线，随后的字符可以是字母、数字或下画线，这也是 C 语言中所有标识符的命名准则。变量的名称不能与 C 语言关键字相同，并且同一个作用域内不能有相同名称的变量。变量的名称与其在程序中的功能毫无关系，但我们在给变量命名的时候应尽量与它的用途联系起来，以利于阅读。

变量可以在定义的时候就给一个初值，称为初始化，如：

```
int count = 0; /* 定义整型变量 count，初值为 0 */
double a = 2.7; /* 定义高精度浮点型变量 a，初值为 2.7 */
```

初始化时所提供的数据类型最好与变量的类型相符。如果不初始化，变量的初值可能自动为 0，也可能是一个不确定的值，故一般情况下应避免去读未初始化的变量。

2.3.2 变量的访问

变量的值可以通过赋值操作符加以改变，也可以作为数据赋值给其他变量，如：

```
int a = 1, b; /* 定义整型变量 a 并初始化为 1，整型变量 b 未初始化 */
a = 2; /* 此行代码执行后，a 的值被改为 2 */
b = a; /* 此行代码执行后，b 的值也为 2 */
```

与初始化相似，通常赋给变量的值也要与变量的类型相符。要注意定义变量时的初始化也用等号表示，虽然很多情况下赋值操作符与它能达到相同的目的，但两者的概念和用法还是有差异的。

2.3.3 左值与右值

进行赋值操作时，变量出现在等号右边和左边有不同的含义：在右边时，变量作为一个数据出现，代表它的值；在左边时，变量作为一个可以存储数据的实体出现，它的值被修改，且与它原来的值无关。习惯上称能够作为数据使用的实体为右值，能够保存数据且保存的数据可以被修改的实体为左值。实质上，左值一定有对应的内存单元，右值则不一定。变量既可以作为左值又可以作为右值，常数则只能作为右值，如：


```
int a;
a = 1; /* a 的值被修改为 1 */
1 = a; /* 语法错误，常数的值不能修改 */
```

2.3.4 只读变量

定义变量时可以使用 `const` 关键字将其指定为只读变量，如：

```
const int m = 100; /* m 定义为只读整型变量 */
int const n = 200; /* int const 和 const int 是相同的 */
```

顾名思义，只读变量只能进行读操作，不能进行写操作，它的值不可以被修改。也就是说，只读变量可以作为右值出现，不能作为左值出现。

2.4 操作符

上文中已提到了赋值操作符，它可以改变一个变量的值。C 语言中的操作符很多，按所需操作数的多少可分为单目操作符、双目操作符和三目操作符；按功能又可分为算术操作符、比较操作符、逻辑操作符、位操作符等。有些操作符会修改操作数本身的值，而另一些则不会，本书中称前者为读写操作符，后者为只读操作符。操作符与操作数作为一个整体又代表一个新的值，称为返回值。一般情况下，返回值只能作为右值使用，不能作为左值使用，本节将要介绍的操作符均满足这一特点。

2.4.1 只读操作符

2.4.1.1 算术操作符

算术操作符的功能与数学上对应的符号类似，它们可以由一个或两个操作数的值进行运算，返回一个新值，新值的类型与原操作数是相同的，如果是双目操作符，则还要求两个操作数的类型相同。C 语言中的算术操作符如表 2.4 所示。

表 2.4 算术操作符

符号	功能	类型	操作数类型	返回值类型
+	正号	单目	整数或浮点数	与操作数相同
-	负号	单目	整数或浮点数	与操作数相同
*	乘法	双目	整数或浮点数	与操作数相同
/	除法	双目	整数或浮点数	与操作数相同
%	模	双目	整数	与操作数相同
+	加法	双目	整数或浮点数	与操作数相同
-	减法	双目	整数或浮点数	与操作数相同

正号和加法虽然用同一个符号表示，但因为一个是单目操作符，一个是双目操作符，语法上不会混淆。负号和减法的情形与此相同。

需要注意的是，因为算术操作符要求结果与原操作数的类型相同，因此当两个整型数做除法操

作时, 返回值也不可能变成小数, 只能是做整除操作。如 $3/4$ 结果是 0 而不可能是 0.75。这时可以通过模操作得到余数, 商乘以除数再加上余数就等于被除数。



模操作用于负整数时, 返回值与所用的计算机平台是有关的, 可能与数学上的概念相异。

2.4.1.2 比较操作符

比较操作符用于对两个同类型操作数的大小进行比较, 返回一个整型值, 这个值为 0 表示判断不成立, 为 1 表示判断成立。C 语言中的比较操作符如表 2.5 所示。

表 2.5 比较操作符

符号	功能	类型	操作数类型	返回值类型
<	小于	双目	整数或浮点数	整型
<=	小于等于	双目	整数或浮点数	整型
>	大于	双目	整数或浮点数	整型
>=	大于等于	双目	整数或浮点数	整型
==	等于	双目	整数或浮点数	整型
!=	不等于	双目	整数或浮点数	整型

使用比较操作符时需要注意, 当操作数是浮点型时, 由于浮点计算的舍入误差, 数学上本来应该相等的数据判断时却可能是不等的, 因此通常把两者差的绝对值是否小于指定的误差范围作为判断浮点数是否相等的依据。



判断相等的操作符 == 很容易误写为赋值操作符 =, 从而导致难以发现的错误。

2.4.1.3 逻辑操作符

C 语言中没有逻辑类型, 而是用整型值来表示逻辑上的“真”与“假”: 0 表示“假”, 非 0 值表示“真”。逻辑操作符对一个或两个整型值进行逻辑运算, 返回一个新的整型值, 用以表示逻辑运算的结果。C 语言中的逻辑操作符如表 2.6 所示。

表 2.6 逻辑操作符

符号	功能	类型	操作数类型	返回值类型
&&	逻辑与	双目	整型	整型
	逻辑或	双目	整型	整型
!	逻辑非	单目	整型	整型

2.4.1.4 位操作符

位操作符可以直接对操作数的各个比特位进行操作。它要求操作数为一个或两个整型值, 返回的结果亦为整型值。C 语言中的位操作符如表 2.7 所示。



表 2.7 位操作符

符号	功能	类型	操作数类型	返回值类型
&	按位与	双目	整型	整型
	按位或	双目	整型	整型
^	按位异或	双目	整型	整型
~	按位取反	单目	整型	整型

2.4.1.5 移位操作符

移位操作符有两种，分别为左移位和右移位，如表 2.8 所示。

表 2.8 移位操作符

符号	功能	类型	操作数类型	返回值类型
<<	左移位	双目	整型	整型
>>	右移位	双目	整型	整型

使用右移位操作符需要注意，它作用于有符号数和无符号数产生的结果是不同的。实际上，计算机内部的右移位操作有两种：算术右移和逻辑右移。算术右移保持操作数的符号位不变，也就是说，对于一个最高位是 1 的数，右移后最高位补的也是 1；而逻辑右移则一律在最高位补 0。C 语言中对移位操作的处理是：有符号数采用算术移位，无符号数采用逻辑移位。



当移位操作符的右侧操作数大于等于 32 时，由于机器指令的局限性，实际移动的位数很可能是该值除以 32 所得的余数。如 `1 << 32` 可能等价于 `1 << 0`，返回值是 1 而不是希望的 0。

2.4.1.6 条件操作符

条件操作符是 C 语言中唯一的三目操作符，其使用方式如下：

```
t ? a : b
```

其中 `t` 为整型值，表示逻辑上的“真”或“假”，`a` 和 `b` 是同类型的数据。如果 `t` 为“真”，则返回的结果就是 `a`，否则就是 `b`。

2.4.1.7 逗号操作符

逗号操作符是双目操作符，它对操作数的类型无要求，返回的结果就是第二个操作数的值，如：

```
1, 2
```

它的返回值就是 2。

2.4.2 读写操作符

一般来说，读写操作符既能改变一个变量的值，又能返回这个变量的值。C 语言中的读写操



作符有以下几类。

一、赋值操作符

赋值操作符是修改变量值的常用形式，它要求左侧的操作数必须是左值，右侧的操作数类型与左侧的操作数类型一致。它的返回值是所赋值的变量的值。

二、自增自减操作符

自增自减操作符都是单目操作符。自增操作符是 `++`，它使操作数的值增加 1；自减操作符是 `--`，它使操作数的值减少 1。当它们置于操作数前面时为前缀方式，这将使所操作变量的值立刻改变，返回值是该变量的值；当它们置于操作数后面时为后缀方式，这时返回值是所操作变量的值，而变量的值直到整个表达式求值完毕后才进行加减。

三、复合赋值操作符

双目的算术操作符、位操作符、移位操作符都有一个对应的复合赋值操作符，如表 2.9 所示。复合赋值操作符可视为它对应的操作符与赋值操作的组合，如：

```
m <<= 3 /* 等价于 m = m << 3 */
```

表 2.9 复合赋值操作符

符号	功能	类型	操作数类型	返回值类型
<code>+=</code>	加后赋值	双目	整数或浮点数	与左侧操作数相同
<code>-=</code>	减后赋值	双目	整数或浮点数	与左侧操作数相同
<code>*=</code>	乘后赋值	双目	整数或浮点数	与左侧操作数相同
<code>/=</code>	除后赋值	双目	整数或浮点数	与左侧操作数相同
<code>%=</code>	模后赋值	双目	整数	与左侧操作数相同
<code>&=</code>	按位与后赋值	双目	整数	与左侧操作数相同
<code> =</code>	按位或后赋值	双目	整数	与左侧操作数相同
<code>^=</code>	按位异或后赋值	双目	整数	与左侧操作数相同
<code><<=</code>	左移后赋值	双目	整数	与左侧操作数相同
<code>>>=</code>	右移后赋值	双目	整数	与左侧操作数相同

它们的返回值是左侧操作数的值。

使用读写操作符的时候要注意，它们的返回值是当用到的时候才从变量中读取的，因此不一定是本操作符赋给变量的值，如：

```
(i = 1) + (++i)
```

得到的值是 4。分析如下：`i` 首先被赋值为 1，随后 `++i` 使 `i` 的值变为 2，到做加法操作时，两个操作数都要读 `i` 此时的值，结果是两个 2 相加等于 4。

2.4.3 类型转换操作符

类型转换操作符由类型名称加上一对圆括号组成，如 `(unsigned int)`，`(float)` 等，作用是将

操作数转换成所指定的类型。它们都是单目操作符，且不会修改操作数。

将取值范围较大的整数类型转换为取值范围较小的整数类型可能会损失数据。将高精度浮点型转换为低精度浮点型，或者将浮点类型与整数类型相互转换，都可能引起转换误差。

在很多情况下，编译器会自动转换操作数的类型。例如，当算术操作符的两侧数据类型不一致时，其中一个操作数的类型会被自动转换为另一个操作数的类型，转换原则是：取值范围较小的整型向取值范围较大的整型转换，整型向浮点型转换，低精度浮点型向高精度浮点型转换。再比如，如果赋值操作符两侧的操作数类型不一致，则右侧数据的类型无条件转换为左侧操作数的类型。



自动转换时，如果有可能损失数据，编译器会给出警告；如果确实需要转换，通过明确地写出类型转换操作符可以消除这一警告。

2.4.4 sizeof 操作符

sizeof 是一个特殊的操作符，它一般只用于变量，返回值是该变量所占内存的大小（字节数）。sizeof 还可以直接对类型名进行操作，返回该类型数据所占的字节数。

事实上，一个变量的类型完全是在编译时确定的，因此不管 sizeof 作用于变量还是类型名，得到的值也一定可以在编译时确定下来，故可以视为一个常数，如：

```
int m, sz;
sz = sizeof(int); /* 等价于 sz = 4 */
sz = sizeof(m); /* 等价于 sz = sizeof(int) */
```

2.5 表达式和语句

2.5.1 表达式

操作符的返回值又可以成为其他操作符的操作数，如此嵌套，就可以形成复杂的表达式。广义地讲，即使只是一个变量名甚至一个常数也是表达式，如：

```
2 /* 常数也是一个表达式，其值为 2 */
m /* 单个变量也是表达式，其值在运行时确定 */
m + n /* 单个操作符与操作数组成的表达式 */
m = n * 3 - 2 /* 多个操作符与操作数组成的表达式 */
```

当表达式中不只一个操作符时，它们的求值顺序由操作符的优先级和结合顺序决定，如表 2.10 所示。表达式中优先级高的操作符先进行求值，如果连续几个操作符的优先级相同，则根据该优先级的结合顺序确定从左还是从右开始求值。

表 2.10 操作符的优先级与结合顺序

操作符	优 级	结 合 顺 序	备 注
++, --	1	从 左 到 右	后缀自增自减

!, ~, ++, --, +, -, 类型转换, sizeof	2	从右到左	单目操作符
*, /, %	3	从左到右	乘除模
(续表)			
操作符	优先级	结合顺序	备注
+, -	4	从左到右	加减
<<, >>	5	从左到右	移位
<, <=, >, >=	6	从左到右	比较操作符
==, !=	7	从左到右	
&	8	从左到右	双目位操作符
^	9	从左到右	
	10	从左到右	
&&	11	从左到右	双目逻辑操作符
	12	从左到右	
?:	13	从右到左	条件操作符
=, +=, -=, *=, /=, %=, &=, ^=, =, <<=, >>=	14	从右到左	赋值与复合赋值操作符

操作符的优先级和结合顺序看似复杂多变，实际上多数是自然形成的。比如为了支持如下的连续赋值操作：

```
a = b = c = 1
```

则赋值操作符的结合顺序必须是从右到左的。再比如单目操作符如果连续作用于一个操作数：

```
(char)!~--++i
```

则单目操作符的结合顺序也必须是从右到左的。

表达式中操作符求值的顺序还可以用括号改变，如：

```
(a + b) * c
```

操作符 + 的优先级比 * 低，但由于括号的作用，运行时将先计算 (a + b)。

2.5.2 语句

表达式后面加一个分号就是一条语句，没有分号则不能单独成为语句。

多条语句连续书写并用大括号括起来称之为复合语句，这些语句将按书写的顺序逐条执行。复合语句的开始处还可以定义变量，如：

```
{  
    int i = 0, j = 0; /* 复合语句开始处可以定义变量 */  
    i + j; /* 无作用，将被编译器优化掉 */  
    i = j + 1;  
}
```

C 语言规定复合语句中的变量只能在开始处定义，即出现在任何非定义变量的语句前面，如下的写法将会产生编译错误：

```
{  
    a = 1;  
    int b; /* 错误，不能在此处定义变量 */  
    b = a;  
}
```



如果语句中的操作符全部是只读的，也就是说，没有对任何变量进行修改，实际上没有产生任何效果，一般情况下会被编译器优化掉。

复合语句还可以再变成表达式，方法是在语句的大括号外面再加上小括号，这时它的值就是最后一条语句的值，例如：

```
int i = ({1; 2; 3;}); /* i 被初始化为 3 */
```

但是这种表达式只能用在函数内部。

2.6 复合类型

本节将介绍由基本数据类型扩展出来的复合数据类型。能够让编程者自定义新的数据类型正是 C 语言强大功能的来源之一。

2.6.1 数组

数组是由多个同类型数据组成的数据类型，其中包含的每一个数据称为数组的元素。数组的元素在内存中是按顺序连续存放的。

2.6.1.1 定义与初始化

定义数组的语法举例如下：

```
int array[5]; /* 定义一个有 5 个元素的数组 array，每个元素都是整型 */
```

这样定义的数组 `array` 可笼统地称为整型数组。实际上，数组的类型有两个要素：数组元素的类型和数组中元素的个数。如果两个数组的元素类型或个数不同，就应被视为不同的数据类型。所以，`array` 的类型确切表达应是“5 个元素的整型数组”，或者用 C 语言自己描述为 `(int [5])` 型。注意数组元素的个数必须是一个常数。



在变量的定义语句中将变量名去掉，剩下的部分就是变量的类型，这是理解 C 语言变量类型的关键。或者说，定义变量的语法就是类型名加上变量名，只不过变量名不一定放在最后。

数组也可以在定义的同时进行初始化，如：

```
int a[5] = {1, 2, 3, 4, 5}; /* a 的 5 个元素的初值分别为 1, 2, 3, 4, 5 */
```

所提供的数据的类型应与数组元素的类型一致。如果数据个数少于数组元素的个数，则没有对应数据的数组元素被初始化为 0。实际上，如果数组在定义的时候同时提供初值，则元素的个数可以省略，如：

```
int b[] = {6, 7, 8}; /* 编译器自动决定 b 的元素个数为 3 */
```

GNU C 还支持另外一种非标准的数组初始化方式，如：

```
char c[] = { /* 省略元素个数，自动决定为数据列表中最大下标加 1，此处为 2 */
    [0] = 'A', /* 第 0 个元素的初值设为字符 A */
    [1] = 'B', /* 第 1 个元素的初值设为字符 B */
}; /* 没有对应数据的数组元素被初始化为 0 */
```

这种初始化方式的好处是，明确地将元素的下标和值对应起来，数据的书写顺序可以任意调换，添加新数据对已有的对应关系没有影响。它还可以与标准方式混用，如：

```
int n[] = {
    [1] = 1, 2, 3, /* 数组 n 的第 1, 2, 3 个元素分别赋初值为 1, 2, 3 */
};
```

2.6.1.2 访问数组元素

一般来说，数组可以保存多个数据，保存的数据可以被修改，数组也对应着一定数量的内存单元，因此数组是左值。但是数组不能直接出现在赋值操作符的左边，原因是赋值操作符不支持数组类型，如下的赋值方法是错误的：

```
int foo[3] = {1, 2, 3}; /* 初始化时可以使用数据列表 */
foo = {4, 5, 6}; /* 语法错误，不能这样赋值 */
```

实际上，赋值操作符只支持基本类型，如果数组元素是基本类型，就可以直接被赋值。访问数组元素的方式是使用 `[]` 操作符，它可以通过数组名称和一个整数值（下标）得到数组的元素，

如：

```
int foo[3] = {1, 2, 3}; /* 定义数组 foo 并初始化 */
int i; /* 定义整型变量 i */
foo[0] = 4; foo[1] = 5; foo[2] = 6; /* 正确的赋值方法 */
i = foo[0]; /* 读数组元素的值, i 的值变为 4 */
foo[i] = 7; /* 下标越界, 可能在运行时崩溃 */
```

下标不仅可以是常数, 也可以是任意变量和表达式的值, C 语言数组的元素下标从 0 开始。
[] 操作符的优先级及结合顺序与后缀自增自减操作符相同。



数组的元素个数在定义时已确定, 但无论是编译时还是运行时都不检查访问数组元素的下标是否在合理的范围内, 因此容易在运行时产生所谓“下标越界”错误, 使程序因访问了非法的内存而崩溃。

2.6.1.3 数组与 sizeof 操作符

当 `sizeof` 操作符作用于数组时, 得到的是数组所占用内存的大小, 如:

```
int array[5]; /* 定义有 5 个元素的整型数组 */
sizeof(int [5]); /* sizeof 直接对类型操作, 值为 5*sizeof(int), 即 20 */
sizeof(array); /* 等价于 sizeof(int [5]) */
sizeof(array[0]); /* array[0] 是 int 型, 故等价于 sizeof(int), 即 4 */
```

因此通过以下方式可以得到数组中元素的个数:

```
sizeof(array)/sizeof(array[0]);
```

2.6.1.4 多维数组

如果数组的每个元素又是数组, 就构成了多维数组, 如:

```
int a[3][4]; /* a 是有 3 个元素的数组, 每个元素都是有 4 个元素的整型数组 */
```

这是一个二维数组, 第一维的元素个数是 3, 第二维的元素个数是 4。它可以用嵌套的数据列表来初始化, 如:

```
int a[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
```

这时要注意, 只有第一维的元素个数才可以省略。

也可以用 GNU C 扩展的初始化方式:

```
int a[][4] = {
    [0] = {
        [0] = 1, /* a[0][0] 的初值设为 1 */
        [1] = 2, /* a[0][1] 的初值设为 2 */
    },
    [1] = {
        [2] = 7, /* a[1][2] 的初值设为 7 */
    },
};
```

```
};
```

这时 `a[0]` 是 `(int [4])` 型的一维数组, 因此不能被赋值, `a[0][0]` 是整型才可以直接被赋值。`a` 的类型可表达为 `(int [3][4])`。`sizeof(a)` 等价于 `sizeof(int [3][4])`, 又等价于 `3*sizeof(int [4])`。

2.6.2 结构体

结构体是由多个不同类型的数据组成的数据类型, 其中的每个数据都要给一个唯一的名称, 称为结构体的成员变量。结构体变量作为一个整体存储在内存中, 它的成员变量之间的相对位置是固定不变的。

2.6.2.1 定义与初始化

结构体变量的定义可以分两步来进行, 先定义新的类型, 再用新类型去定义变量。定义结构体类型使用关键字 `struct`, 如:

```
struct my_struct { /* 结构体的名称为 my_struct */
    char ch;
    int n;
};
```



定义结构体类型时, 最后的分号不可缺少。

这样就定义了一个新的结构体数据类型, 它包含一个字符型成员 `ch` 和一个整型成员 `n`。定义的新类型的名称为 `struct my_struct` (必须写上 `struct` 关键字), 可以用新类型来定义变量, 如:

```
struct my_struct var; /* 定义 struct my_struct 型的变量 var */
```

也可以在定义类型的同时定义变量, 如:

```
struct my_struct {
    char ch;
    int n;
} var; /* 既定义了类型 struct my_struct, 又定义了变量 var */
```

这时结构体的名称还可以省略, 但是这样它就成了无名类型, 编译器会自动为其产生一个内部使用的名称, 这个名称在编写程序时是无法知道的, 因此就不能用它再定义别的变量。

结构体的名称和定义的变量名可以相同, 如:

```
struct my_struct my_struct; /* 变量名为 my_struct */
```

因为结构体类型名总是带着 `struct` 关键字, 因此不会混淆。

定义结构体变量时, 也可以同时初始化, 如:

```
struct my_struct var = {'A', 5}; /* var 的成员 ch 初始化为字符 A, n 为 5 */
```

初始化所提供的数据列表中的数据类型应与结构体中相应成员的类型相匹配。

对于结构体，GNU C 也支持一种扩展的初始化方式，如：

```
struct my_struct var = {
    .ch = 'A', /* 成员 ch 初始化为字符 A，注意成员名前的小数点 . */
    .n = 5, /* 成员 n 初始化为 5 */
}; /* 没有提供数据的成员将被初始化为 0 */
```

这种初始化方式与数组初始化的 GNU C 扩展方式有相同的优点。

2.6.2.2 访问结构体成员

结构体变量也是左值，对结构体变量进行直接赋值是允许的，这时要求赋值表达式的右侧也是相同类型的结构体，如：

```
struct my_struct var1 = {'A', 5};
struct my_struct var2;
var2 = {'A', 5}; /* 语法错误，数据列表只能用在初始化时 */
var2 = var1; /* var1 的每个成员的值赋给 var2 的对应成员 */
```



并不是所有的编译器都支持这种直接赋值的方法，因此应尽量避免使用。

从结构体变量可以访问到它的每一个成员，这时要用到 `.` 操作符，如：

```
struct my_struct var;
var.ch = 'A'; /* var 的成员 ch 赋值为字符 A */
var.n = 5; /* var 的成员 n 赋值为 5 */
```

`.` 操作符的优先级及结合顺序与 `[]` 操作符相同。

2.6.3 位域

如果一个数据的取值范围很小，比如逻辑型数据只能取 0 和 1 两个值，这时候用一个字节来表示就存在浪费。在结构体中，可以使用位域的方法来指定数据所占用的比特数，其语法举例如下：

```
struct bit_field {
    char a:4; /* 成员 a 定义为 char 型，但只占 4 个比特 */
    char b:4; /* 成员 b 定义为 char 型，但只占 4 个比特 */
}; /* 成员 a 与 b 放在同一个字节中，占 8 个比特 */
```

使用位域的成员必须定义为整数类型。一般来说，希望组合在同一个整数类型数据中的位域成员就应该定义成这种类型。下面这种位域的定义方式与上述方式等价：

```
struct bit_field {
    char a:4, b:4; /* 成员 a 和 b 定义为 char 型，各占 4 个比特 */
}; /* 形式上 a 和 b 有 char 型数据的“一部分”的感觉 */
```

如果定义成这样:

```
struct bit_field {
    char a:4; /* 成员 a 定义为 char 型, 但只占 4 个比特 */
    short b:4; /* 成员 b 定义为 short 型, 但只占 4 个比特 */
}; /* 成员 a 与 b 放在同一个 short 型数据中, 占 16 个比特 */
```

这时虽然成员 a 和 b 加起来只有 8 位, 但由于 b 被定义为 short 型, 所以编译器为它们分配了一个 short 的空间。再看下面的例子:

```
struct bit_field {
    char a:4; /* 成员 a 定义为 char 型, 但只占 4 个比特 */
    char b:5; /* 成员 b 占 5 个比特, 不可能与 a 放在同一个字节内 */
}; /* 成员 a 与 b 各放在一个字节中 */
```

编译器会保证一个成员的所有比特位放在同一个数据中, 如果正在分配的数据空间已容纳不了下一个成员, 编译器将分配一个新的数据。比特数为 0 的成员可以让编译器立刻启用一个新的数据空间, 如:

```
struct bit_field {
    char a:4; /* 成员 a 定义为 char 型, 只占 4 个比特 */
    char :0; /* 比特位为 0 的成员没有实际用途, 因此名称可省略 */
    char b:4; /* 从这里开始使用一个新的字节 */
}; /* 成员 a 与 b 各放在一个字节中 */
```



多数编译器都将先定义的位域成员放在数据的低位上, 但也有一些编译器的处理是相反的, 将先定义的位域成员放在高位上。

2.6.4 数据的对齐

由于硬件设计的特点, CPU 在读写内存时, 如果所给的地址是机器字长的整数倍, 则操作效率会比较高, 这可以称为地址的对齐。有些 CPU 甚至根本不支持读写地址不对齐的内存单元, 这时需要用软件来模拟, 操作的效率就更低了。在一般的 32 位机器上, 地址对齐的界线是 4 的整数倍。

有鉴于此, 为了提高程序运行效率, 编译器会尽量避免一个变量 (包括结构体成员) 的存储空间跨越地址对齐的界线。比如以下的结构体定义:

```
struct my_struct {
    char ch1; /* 字符型成员 ch1 占据 1 个字节 */
    char ch2; /* 字符型成员 ch2 占据 1 个字节 */
    int n; /* 整型成员 n 占据 4 个字节 */
    char ch3; /* 字符型成员 ch3 占据 1 个字节 */
}; /* 整个结构体可能占据 12 个字节 */
```

如果要求地址与 4 的整数倍对齐, 则这个结构体成员的存储空间分配如图 2.1 所示。成员 ch1 和 ch2 放在同一个对齐单元中, 虽然它们只占两个字节, 但为了让成员 n 的存储空间不跨越对齐界线, 需要补充两个空闲的字节。成员 ch3 后尽管已经没有其他成员, 但仍然需要补充三

个空闲字节，使整个结构体的长度达到 4 的整数倍，这是因为如果结构体组成数组，则要求每个数组元素的地址都是对齐的。

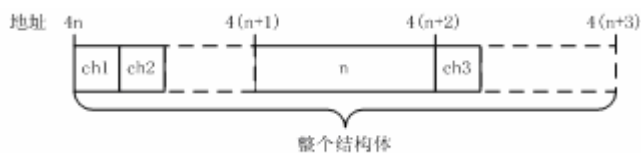


图 2.1 结构体成员的对齐

对结构体类型使用 `sizeof` 操作符，得到的是结构体占用的内存字节数，包括所有的空闲字节，显然，这个值并不等于它的所有成员的大小之和。

在很多情况下，结构体中不允许有空闲字节，所有成员必须一个紧挨一个地存放，或者说，地址的对齐边界是 1 的整数倍。这时可以用如下预处理语句来指定：

```
#pragma pack(1) /* 将地址对齐界线改为 1 的整数倍 */
struct my_struct {
    char ch1; /* 字符型成员 ch1 占据 1 个字节 */
    char ch2; /* 字符型成员 ch2 占据 1 个字节 */
    int n; /* 整型成员 n 占据 4 个字节 */
    char ch3; /* 字符型成员 ch3 占据 1 个字节 */
}; /* 整个结构体占据 7 个字节 */
#pragma pack() /* 将地址对齐界线改回原来的值 */
```

GNU C 支持结构体的扩展属性，也可以做到这一点，如：

```
struct my_struct {
    char ch1; /* 字符型成员 ch1 占据 1 个字节 */
    char ch2; /* 字符型成员 ch2 占据 1 个字节 */
    int n; /* 整型成员 n 占据 4 个字节 */
    char ch3; /* 字符型成员 ch3 占据 1 个字节 */
} __attribute__((packed)); /* 指明此结构体是“紧凑”的，占据 7 个字节 */
```

2.6.5 联合体

联合体的定义和使用与结构体类似，只是关键字由 `struct` 换成了 `union`，如：

```
union my_union { /* 联合体的名称为 my_union */
    char ch;
    int n;
}; /* 定义了联合体类型 union my_union */
union my_union var; /* 定义 union my_union 型变量 var */
```

联合体和结构体的区别在于，结构体的各个成员都有自己独立的存储空间，而联合体各个成员的存储空间相互重叠，都从同一内存地址开始。初始化时，联合体变量被认为只有一个成员，如：

```
union my_union var = {'A', 10}; /* 成员 ch 的值赋为字符 A，10 被忽略 */
```

修改联合体变量的一个成员，则其他的成员也被修改，如：

```
var.n = 1000;
var.ch = 'A'; /* 执行这条语句后, var.n 的值也变了, 一般情况下是 833 */
```

这可以由图 2.2 解释。字符 A 的 ASCII 码值为 0x41, 对 var.ch 赋值以后, 实际上等价于把 var.n 的第一个字节改成了 0x41。

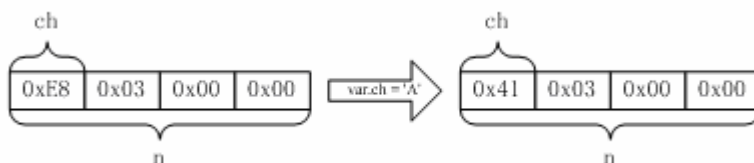


图 2.2 联合体成员赋值图解

对联合体类型进行 `sizeof` 操作, 得到的结果是其中最长的一个成员的长度。

使用联合体的目的就是为了解约内存, 但前提是它的各个成员不能同时有效。形象地说, 结构体相当于多个数据的“串联”, 联合体相当于多个数据的“并联”。

2.6.6 类型嵌套

结构体、联合体类型都可以组成数组, 它们的成员也可以是数组。结构体、联合体之间也可以相互嵌套, 如:

```
struct B {
    int n;
};

struct A {
    struct B b; /* 成员 b 是 struct B 型 */
} var; /* 定义了 struct A 型的变量 var */
```

访问变量 `var` 的成员 `b` 的成员 `n` 时, 要使用两次 `.` 操作符, 如:

```
var.b.n = 100;
```

也可以直接在结构体内定义另外一个结构体类型及这种类型的成员, 如:

```
struct A {
    struct B { /* 定义结构体 B */
        int n;
    } b; /* 定义 struct B 型的成员 b */
} var;
```

GNU C 支持一种扩展的嵌套类型定义, 结构体 (或联合体) 内部定义的结构体 (或联合体) 可以既不给类型名又不给成员变量名, 如:

```
struct A {
    int type;
    union { /* 此处没有类型名 */
        char ch;
```

```

    int n;
}; /* 此处没有成员变量名 */
} var;

```

这时内部的 `union` 关键字其实没有定义新的类型，它的含义是成员 `ch` 和成员 `n` 仍是结构体 `A` 的成员，但以联合体的方式处理，即占用相同的存储空间，如：

```

var.type = 1; /* 成员 type 占据单独的存储空间 */
var.ch = 'A'; /* 成员 ch 赋值为字符 A */
var.n = 100; /* 对成员 n 赋值，结果成员 ch 的值也变了 */

```

这种语法可以用来构造成员之间既有“串联”又有“并联”的复杂数据结构。需要注意的是，这种用法并不属于 C 语言标准。

2.6.7 类型别名

用 `typedef` 关键字可以建立类型的别名，如：

```

typedef struct my_struct {
    int m, n;
} my_struct; /* GNU C 允许新类型名与结构体名相同 */

```

这些语句定义了一个新类型 `struct my_struct`，并且给它定义了一个别名 `my_struct`。此后 `my_struct` 就可以代替 `struct my_struct` 作为类型的名称去定义变量了，如：

```

my_struct a, b; /* 定义 my_struct 型变量 a 和 b */

```

`typedef` 可以给任何已定义的类型定义别名，包括各种基本类型，如：

```

typedef unsigned long Ulong; /* 此后 Ulong 就代表无符号长整型 */
typedef unsigned char Uchar; /* 此后 Uchar 就代表无符号字符型 */

```



`typedef` 的语法可以这样理解，如果去掉 `typedef` 关键字，这条语句就成了定义变量的语句，所定义的新类型名就成了变量名。

2.6.8 枚举类型

枚举类型通常用来表示只有几个可选值的数据类型，可用如下的语法定义：

```

enum num {
    ONE = 1, /* 如果不指定将从 0 开始 */
    TWO, /* 自动定义为上一个值加 1，即 2 */
    THREE, /* 值为 3，留一个逗号方便以后添加 */
};

```

这样，`enum num` 就成了一个新的类型名，可以用来定义变量：

```

enum num a = ONE;

```

实际上，编译器会把枚举类型处理为整型，因此这条语句等价于：

```
int a = 1;
```

枚举类型通常用来批量定义整型常数。

2.7 流程控制

根据结构化编程的原则，程序的流程基本可分为三类：顺序结构、分支结构和循环结构。在 C 语言中可以方便地写出这几种结构的代码，也可以实现执行流程的无条件转移。

2.7.1 顺序结构

顺序结构是最简单的程序结构，运行时从上到下顺序执行各条语句。顺序结构不需要任何附加的语法元素，只需按顺序书写各条语句即可，举例如下：

```
{
    int i = 0, n[3] = {0, 0, 0};
    n[i++] += i; /* n[0] 的值增加 0, 然后 i 的值加 1 */
    n[i++] += i; /* n[1] 的值增加 1, 然后 i 的值加 1 */
    n[i++] += i; /* n[2] 的值增加 2, 然后 i 的值加 1 */
}
```

2.7.2 分支结构

分支结构的特点是可以根据某个条件的成立与否去执行不同的语句。C 语言中的分支结构可分为两种：if 语句和 switch 语句。需要注意的一点是，即使没有出现这些语句，在表达式中也可能隐含有分支结构。

2.7.2.1 if 语句

if 语句的语法如下：

```
if (cond) s1; else s2;
```

其各个部分的含义解释如下。

- ◆ cond：表达式，表示要判断的条件。
- ◆ s1, s2：语句，可以是复合语句。

整个语句的作用是：当 cond 的值非 0 时，执行语句 s1；当 cond 的值为 0 时，执行语句 s2。这里的 else 子句可以省略。

if 语句的使用举例如下：

```
if (a < 0) a = -a; /* 如果 a < 0, a 的值变为它的相反数, 否则无操作 */
```

另一个例子：

```
if ('a' <= ch && ch <= 'z') { /* 如果 ch 是小写字母 */
```



```
    ch += 'A'-'a'; /* 将它变为大写字母 */
} else { /* 否则 */
    ch = '-'; /* 将它变为一个减号 - */
}
```

if 语句可以嵌套，即它的某个子句中还可以包含 if 语句。这时要注意 else 和 if 的匹配问题。如下是一段用于成绩分类的代码：

```
pass = 1; /* 置 pass 为 1，表示成绩默认是及格的 */
if (score >= 60) /* 如果成绩超过 60 */
    if (score >= 80) class = 'A'; /* 如果成绩又超过 80 就是 A 类 */
else /* 否则 */
    pass = 0; /* 置 pass 为 0，表示不及格 */
```

代码的初衷是：成绩超过 60 的为及格，小于 60 的为不及格，成绩超过 80 的标记为 A 类。但实际上，由于 else 子句总是跟离它最近的 if 匹配，代码的功能变成：分数小于 60 的为及格，分数超过 60 但小于 80 的反而为不及格。为使代码的流程清晰，必须加上大括号：

```
pass = 1; /* 置 pass 为 1，表示成绩默认是及格的 */
if (score >= 60) { /* 如果成绩超过 60 */
    if (score >= 80) class = 'A'; /* 如果成绩又超过 80 就是 A 类 */
} else { /* 否则 */
    pass = 0; /* 置 pass 为 0，表示不及格 */
}
```

2.7.2.2 switch 语句

switch 语句的语法如下：

```
switch (exp) {
case v1: s1;
case v2: s2;
default: sd;
}
```

其各个部分的含义解释如下。

- ◆ exp：必须是整数类型的表达式。
- ◆ v1, v2：与 exp 同类型的表达式，如果类型不同，可能发生自动转换。
- ◆ s1, s2, sd：空语句、一条语句或多条语句。

整个语句的作用是：当 `exp == v1` 时，流程跳转到 s1 执行；当 `exp == v2` 时，流程跳转到 s2 执行；否则，流程跳转到 sd 执行。

switch 语句的典型用法举例如下：

```
switch (mark) {
case 'A':
case 'B':
    class = 1;
}
```

```

    break; /* break 语句跳出 switch 结构 */
case 'C':
    class = 2;
    break; /* break 语句跳出 switch 结构 */
default:
    class = 3;
}

```

使用 `switch` 语句时需要注意：`case` 及 `default` 子句只起一个类似标号的作用，当流程跳转到某个 `case` 或 `default` 子句后，会一直向下顺序执行所有的语句，不管中间有没有经过其他 `case`。这时要利用 `break` 语句，它可以无条件地从 `switch` 结构中跳出来，使流程转移到 `switch` 结构之后。

`switch` 语句的功能完全可以用多个嵌套的 `if` 语句来模拟。如上述的 `switch` 结构功能上等价于以下 `if` 结构：

```

if (mark == 'A' || mark == 'B') {
    class = 1;
} else if (mark == 'C') {
    class = 2;
} else {
    class = 3;
}

```

2.7.3 隐含的分支结构

在 C 语言中，有三种操作符实质上隐含了分支结构。

一、条件操作符

条件操作符本身隐含了分支结构，如：

```
n = (cond) ? exp1 : exp2;
```

这里要注意的是：当 `cond` 不为 0 时，只对 `exp1` 求值，不对 `exp2` 求值；当 `cond` 为 0 时，不对 `exp1` 求值，只对 `exp2` 求值。

上述语句的功能实际上等价于：

```
if (cond) n = exp1; else n = exp2;
```

二、逻辑与操作符

逻辑与操作符本身隐含了分支结构，如：

```
if (cond1 && cond2) s;
```

这里要注意的是：当 `cond1` 不为 0 时，继续对 `cond2` 求值；当 `cond1` 为 0 时，不对 `cond2` 求值。

上述语句的功能实际上等价于：

```
if (cond1) if (cond2) s;
```

三、逻辑或操作符

逻辑或操作符本身隐含了分支结构，如：

```
if (cond1 || cond2) s;
```

这里要注意的是：当 `cond1` 不为 0 时，不对 `cond2` 求值；当 `cond1` 为 0 时，继续对 `cond2` 求值。

上述语句的功能实际上等价于：

```
if (cond1) s; else if (cond2) s;
```

2.7.4 循环结构

循环结构的特点是可以重复执行某段代码，C 语言中的循环结构可分为三种：`while` 语句、`for` 语句和 `do while` 语句。

2.7.4.1 while 语句

`while` 语句的语法如下：

```
while (cond) s;
```

其各个部分的含义解释如下。

- ◆ `cond`：表达式，循环的终止条件。
- ◆ `s`：语句，可以是复合语句，也可以是空语句，称为循环体。

这条语句的作用是：当 `cond` 的值非 0 时，执行语句 `s`，执行完毕后流程返回 `while` 语句的开始，再次对 `cond` 求值并判断，如此周而复始；当 `cond` 的值为 0 时，循环结束，不执行语句 `s`，流程跳转到整个 `while` 结构之后。

`while` 语句的使用举例如下：

```
while (i-- > 0); /* 当 i 的值不大于 0 时循环结束，否则 i 的值不断减 1 */
```

另一个例子：

```
i = sizeof(a)/sizeof(a[0])-1; /* i 的初值赋为数组 a 中最后一个元素的下标 */
while (i > 0) {
    a[i] = a[i-1];
    i--;
} /* 将数组 a 的元素（除最后一个）的值依次向后移动一个位置 */
```

2.7.4.2 for 语句

`for` 语句的语法如下：

```
for (s1; cond; s2) s;
```

其各个部分的含义解释如下。

- ◆ cond: 表达式, 循环的终止条件。
- ◆ s1: 循环的初始化语句, 可以为空。
- ◆ s2: 语句, 可以为空。
- ◆ s: 语句, 可以为空或复合语句, 称为循环体。

这条语句的作用是: 首先执行语句 **s1**, 然后开始循环; 当 **cond** 的值非 0 时, 执行语句 **s**, 再执行语句 **s2**, 流程返回到判断 **cond** 处, 如此周而复始; 当 **cond** 的值为 0 时, 循环结束, 流程跳转到整个 **for** 结构之后。实际上等价于:

```
s1; while (cond) {s; s2; }
```

for 语句的用法举例如下:

```
for (i = 0; i < sizeof(a)/sizeof(a[0]); i++) {
    a[i] = 0;
} /* 将数组 a 的元素全部赋值为 0 */
```

另一个例子:

```
for (i = 0; i < sizeof(a)/sizeof(a[0]) && a[i] != 0; i++);
```

此循环结束后, **i** 的值是数组 **a** 中第一个值为 0 的元素的下标, 如果数组 **a** 的元素都不是 0, 则 **i** 的值是数组中元素的个数。

2.7.4.3 do while 语句

while 语句是先判断条件再执行循环体, 而 **do while** 语句则是先执行循环体再判断条件, 其语法如下:

```
do s; while (cond);
```

其各个部分的含义解释如下。

- ◆ s: 语句, 可以是复合语句, 称为循环体。
- ◆ cond: 表达式, 循环的终止条件。

这条语句的作用是: 先执行语句 **s**, 然后进行判断, 当 **cond** 的值非 0 时, 流程返回开头, 如此周而复始; 当 **cond** 的值为 0 时, 循环结束, 流程跳转到整个 **do while** 结构之后。

do while 语句的用法举例如下:

```
do {
    m = 0;
    n = 0;
} while (0); /* 即使条件不成立, 循环体也要执行一次 */
```

2.7.4.4 break 和 continue

break 可以用来跳出 **switch** 结构, 也可以用来跳出循环结构, 包括 **while** 结构、**for** 结构

和 `do while` 结构，如：

```
for (sum = 0, i = 0; i < sizeof(a)/sizeof(a[0]); i++) {
    if (a[i] == 10) break; /* 如果 a[i] 为 10 则跳出循环 */
    sum += a[i];
} /* 该循环将数组 a 中第一个值为 10 的元素前的所有元素的和存入 sum 变量 */
```

`continue` 语句用在循环体中，可以使流程立刻跳转到下一轮循环的开始，如：

```
for (sum = 0, i = 0; i < sizeof(a)/sizeof(a[0]); i++) {
    if (a[i] == 10) continue; /* 如果 a[i] 为 10 则开始下一轮循环 */
    sum += a[i];
} /* 该循环将数组 a 中所有值不为 10 的元素的值的和存入 sum 变量 */
```

实际上等价于：

```
for (sum = 0, i = 0; i < sizeof(a)/sizeof(a[0]); i++) {
    if (!(a[i] == 10)) {
        sum += a[i];
    }
} /* 该循环将数组 a 中所有值不为 10 的元素的值的和存入 sum 变量 */
```

可见，恰当使用 `continue` 语句的好处是可以减少代码的缩进量，使流程更清晰。

`break` 和 `continue` 语句都只对它所在的循环起作用。例如，当 `break` 处在多层嵌套循环的最内层时，它只能使流程跳转到最内层循环外，而不能直接跳出外层循环。

2.7.5 goto

`goto` 是无条件跳转语句，可以跳转到代码中任何有标号的地方，如：

```
i = 0;
begin: /* 标号，形式是标识符加一个冒号 : */
    if (i >= sizeof(a)/sizeof(a[0])) /* 如果条件成立 */
        goto end; /* 流程跳转到 end 标号处 */
    a[i] = 0;
    i++;
    goto begin; /* 流程跳转到 begin 标号处 */
end: /* 标号 */
```

这段代码实际上模拟了如下的循环结构：

```
for (i = 0; i < sizeof(a)/sizeof(a[0]); i++) a[i] = 0;
```

`goto` 可以用来从多层嵌套循环的内层循环体中直接跳转到外层循环后，如：

```
for (i = 0; i < sizeof(a)/sizeof(a[0]); i++) {
    for (j = 0; j < sizeof(a[0])/sizeof(a[0][0]); j++) {
        if (a[i][j] == 0) goto loop_end; /* 条件成立则跳转到 loop_end */
    }
}
loop_end:
```

这段代码的功能是查找二维数组中第一个为 0 元素的下标。

定义标号时要注意，标号后面不能直接是复合语句结束的大括号，需要的话可以在标号后面加一个空语句解决这个问题。



因为无条件跳转会使代码的可读性变差，所以不能滥用 goto，如果能使用基本的分支结构、循环结构，则尽量不要使用 goto。

2.8 函数

C 语言是一种“函数式”的语言，所有的可执行代码都放在函数里。函数之间可以相互调用，形成复杂的执行流程。

2.8.1 声明与定义

2.8.1.1 函数的定义

定义函数的语法如下：

```
type name(type1 arg1, type2 arg2)
{
    s;
}
```

其各个部分的含义解释如下。

- ◆ type: 函数的返回值类型。
- ◆ name: 函数的名称。
- ◆ type1, type2: 参数的类型。
- ◆ arg1, arg2: 参数的名称。
- ◆ s: 调用函数时要执行的复合语句，称为函数体。

函数的定义举例如下：

```
int sum(int a, int b)
{
    return a+b; /* 返回 a 与 b 的和 */
}
```

这个函数用来求两个数的和，其中 return 语句表示函数执行到这里结束，并返回一个数值。函数也可以不返回任何值，这时要用关键字 void 来定义，如：

```
void do_nothing(void) /* void 也可以用在参数列表处，表示没有参数 */
{
    return; /* return 不返回任何值 */
} /* 此函数不做任何操作 */
```

函数的定义不可以嵌套，也就是说，不能在函数体内定义函数。

2.8.1.2 函数体内的流程控制

函数体内可以使用各种流程控制结构，如顺序结构、分支结构和循环结构等。使用时要注意以下两个关键字的作用：**return** 和 **goto**。

一、return

如果没有 **return** 语句，则函数会一直执行到达最后的大括号才返回。使用 **return** 语句使函数返回实际上有流程控制的作用，如：

```
int absolute(int n) /* 求一个整数的绝对值 */
{
    if (n >= 0) return n; /* 如果 n 是正数或 0，直接返回 n */
    n = -n; /* 如果 n 是正数或 0，这里就执行不到 */
    return n;
}
```

实际上等价于：

```
int absolute(int n) /* 求一个整数的绝对值 */
{
    if (!(n >= 0)) {
        n = -n;
    }
    return n;
}
```



如果定义为有返回值的函数，但函数体中没有 **return** 语句，则返回值是不定的，通常这种情况编译器会给出警告。

二、goto

函数体的代码具有整体性，不能使用 **goto** 语句跳转到函数体外的任何地方，也就是说，不能跳转到其他函数内。只有当函数被调用时，函数体中的代码才有可能被执行。

由于 **goto** 语句的作用被局限在一个函数内，因此在不同函数中可以使用相同的标号而不会产生混淆，但同一个函数内的标号必须唯一。

2.8.1.3 函数的声明

如果将定义函数的整个函数体去掉，并以分号结束，就成了对函数的声明，如：

```
int sum(int a, int b); /* 声明函数 sum */
```

声明函数的目的是让编译器知道函数的类型，函数的类型由它的返回值类型及参数的个数与类型决定。为了与函数的返回值类型相区别，函数本身的类型通常称为函数的原型。

调用函数时，只需要知道函数的原型就可以了，不需要知道函数的整个定义。正因为如此，才可以把一个程序的源代码写在多个源文件中，一个源文件中可以调用另一个源文件中的函数。

在同一个程序中，一个函数可以多处声明，但只能有一个定义。一个函数在多处声明时必须要有相同的原型。

声明函数时的形参名可以省略，如：

```
int sum(int, int); /* 形参省略 */
```

此函数的原型可表达为 (int (int, int)) 型。



明确地写出形参名可以让代码读者“望文知义”，起到增强可读性的作用。

2.8.2 函数的调用与传值方式

可以通过函数名调用函数，如：

```
int n = sum(2, 3); /* n 得到 sum(2, 3) 的返回值 */
```

函数的调用过程可做如下解释。

- step 1 表达式中出现函数，必须调用函数进行求值。
- step 2 执行流程进入函数体内，各个参数取得调用者提供的对应值。
- step 3 执行函数体内的语句，直到语句结束或碰到 **return** 语句。
- step 4 函数执行完毕，**return** 语句中表达式的值就是函数的返回值。
- step 5 执行流程回到调用函数处，函数的返回值在表达式内得到使用。

定义函数时所使用的参数名并没有具体的值，称为形式参数，或简称为形参；调用函数时小括号内提供的数据列表称为实际参数，或简称为实参。形参在函数体内可作为变量使用，这个变量的初值是调用时提供的对应的实参值。如果对应的形参和实参类型不匹配，则编译器会将实参的值自动向形参的类型转换。

调用函数时需要注意，函数的返回值均不能作为左值使用，无返回值的函数不能用在其他表达式中。

C 语言支持函数的递归调用。递归调用的最简单形式是在函数体内调用自己，如：

```
int factorial(int n) /* 此函数用于求阶乘 */
{
    if (n == 1) return 1; /* 递归终止条件 */
    return n*factorial(n-1); /* n! 等于 n*(n-1)! */
}
```

函数的相互调用也可以形成递归，比如在函数 A 内调用了函数 B，而在函数 B 内又调用了函数 A。



每次调用函数时将在程序的运行栈上消耗存储空间，如果调用层次过多，将耗尽栈上的空间而导致程序崩溃。因此使用递归时应小心设置终止条件，否则极易形成无限次嵌套调用。



调用函数时需要正确理解函数参数的传值方式。例如，试图通过以下的函数交换两个变量的值是行不通的：

```
void swap(int a, int b)
{
    int t; /* 定义临时变量 t 用于暂存数据 */
    t = a; a = b; b = t; /* 参数 a 和 b 的值被交换 */
}
```

调用时：

```
int a = 1, b = 2;
swap(a, b); /* 实际上传递的是 a 和 b 的值，等价于 swap(1, 2) */
```

调用完成后 a 和 b 的值不受影响。

2.8.3 函数与复合类型

函数的参数和返回值都不能是数组，但可以是结构体或联合体，例如：

```
struct my_time {
    int hour, minute, second;
}; /* 结构体 my_time 有三个成员，分别表示时、分、秒 */

struct my_time get_time(int second) /* 将秒数转换成时、分、秒 */
{
    struct my_time tmp; /* 临时变量，用于暂存时、分、秒的数值 */
    tmp.minute = second / 60; /* 分钟数等于秒数整除以 60 */
    tmp.second = second % 60; /* 余数是最终的秒数 */
    tmp.hour = tmp.minute / 60; /* 小时数是分钟数整除以 60 */
    tmp.minute %= 60; /* 余数是最终的分钟数 */
    return tmp; /* 返回结构体 */
}
```

调用时：

```
struct my_time t = get_time(sec); /* 把返回的结构体赋值给另一个结构体 */
int hour = get_time(sec).hour; /* 可以取返回的结构体的成员值 */
```

当参数是结构体（或联合体）时，实参的每个成员的值将传递给形参的对应成员；当返回值是结构体（或联合体）时，可以把它赋值给另外一个同类型的结构体（或联合体），也可以取它的成员的值。

实际上很少直接使用结构体作为参数和返回值，因为调用时的参数传递开销会很大。

2.8.4 内联函数

GNU C 支持用 `inline` 关键字把函数定义为内联函数，如：

```
static inline sum(int a, int b)
{
```



```
return a+b;
}
```

内联函数的功能与普通函数相同,但是当打开编译器的优化选项时,内联函数实质上不生成独立的代码,而是将代码直接嵌入到调用它的地方,因此可以省去函数的调用开销,加快执行速度。

即使在打开编译器优化选项的情况下,由于函数本身和使用上的一些原因,内联函数也可能被当做普通函数来编译。为了使内联函数产生效果,一般需要将 **inline** 关键字与 **static** 关键字一起使用。同时,内联函数每被调用一次,就相当于函数体的代码重写一遍,因此函数体应尽量短小。

注意内联函数并不属于 C 语言标准。

2.8.5 变量的作用范围与生存期

2.8.5.1 局部变量和全局变量

在复合语句的开始处定义的变量称为局部变量,这种变量只能在这个复合语句内部使用。可以合法地使用某个变量的代码范围称为这个变量的作用范围。换句话说,局部变量的作用范围从定义后开始,到复合语句的末尾结束。

函数体是一个复合语句,因此函数内定义的变量是局部变量。

变量还可以定义在函数外面,称为全局变量。全局变量的作用范围自定义后开始,到整个源文件末尾结束。也就是说,全局变量在它定义之后的各个函数内都可以使用。

全局变量的作用范围还可以跨越文件。例如,在源文件 **a.c** 中定义一个全局变量:

```
/* 文件名: a.c */
int a; /* 定义全局整型变量 a */
```

在源文件 **b.c** 中使用这个变量:

```
/* 文件名: b.c */
extern int a; /* 声明 a 为外部的整型变量 */
int main(void)
{
    a = 0; /* 对 a 赋值 */
    return 0;
}
```

因为 C 语言的各个源文件是单独进行编译的,所以使用其他文件内的全局变量必须用 **extern** 关键字进行声明。与函数一样,全局变量可以有多个声明,但只能有一个定义。

事实上, **extern** 关键字也可用于声明外部函数,只不过因为函数的声明默认就是外部的,所以 **extern** 关键字可以省略。

在变量的作用范围之外可以定义与它同名的变量,两者互不干扰。如果定义的局部变量与全局变量或者上一层复合语句中的局部变量同名,则存在作用范围的覆盖问题,例如:

```
int i; /* 定义全局变量 i */
void fun(void)
{
    int i; /* 定义局部变量 i */
    i = 1; /* 实际访问的是局部变量 i */
}
```

}

覆盖的原则是：作用范围小的变量取代作用范围大的变量，也就是说，内层复合语句中的局部变量取代外层复合语句中的同名局部变量，局部变量取代同名的全局变量。

2.8.5.2 静态变量

变量的作用范围是一个编译时的概念。在程序运行时，变量还有生存期的概念。一般的局部变量当程序执行到它的定义语句时才产生，执行到它所在的复合语句结束时消失，这就是它的生存期，这种变量又称为自动变量。自动变量的值只在生存期内保持，下一次产生时将重新被初始化（如果不初始化则初值是不确定的），例如：

```
int fun(void)
{
    int i = 0; /* i 是自动变量，每次调用 fun 函数，i 的初值都是 0 */
    i++;
    return i; /* 返回值永远是 1 */
}
```



自动变量都放在程序的运行栈上，它所对应的内存单元是调用到所在函数时才分配的，两次函数调用时给同一自动变量分配的内存单元可能不同。

与此相反，全局变量是程序开始运行时就存在的，直到程序运行结束才消失。对全局变量的初始化只在程序开始运行时执行一次。

定义局部变量时，可以用 **static** 修饰符改变它的生存期，这样定义的变量称为静态变量，例如：

```
int fun(void)
{
    static int i = 0; /* i 是静态变量，只在程序开始运行时初始化一次 */
    i++;
    return i; /* 第一次调用 fun 将返回 1，第二次返回 2 */
}
```

静态变量的生存期与全局变量相同，因此，静态变量只在程序开始运行时初始化一次，函数中的静态变量在下次函数被调用时保持上一次调用结束时的值。

将全局变量定义为静态变量，实际上改变的是它的作用范围，这样定义的变量作用范围缩小至本源文件内，在其他源文件中即便用了 **extern** 关键字进行声明也引用不到这个变量；同时，其他源文件中就可以合法地定义同名变量而不产生冲突。

static 关键字也可用于声明函数，例如：

```
static void fun(void); /* 声明 fun 为 static 函数 */
```

它将使函数的作用范围缩小至本源文件内，这样在其他源文件中就不可能调用这个函数；同时，其他源文件中就可以定义与它同名的函数而不产生冲突。一般来说，如果一个函数只在本源文件中使用，最好加上 **static** 关键字。



对一个函数只需要使用一次 `static` 关键字，如果声明函数时使用了 `static` 关键字，则定义时就可以省略。

如表 2.11 所示是对变量作用范围与生存期的总结。

表 2.11 变量的作用范围与生存期

定义的位置	修饰	作用范围	生存期
局部变量	无	仅在本复合语句内	仅在执行本复合语句时
局部变量	静态	仅在本复合语句内	运行时一直存在
全局变量	无	所有源文件内	运行时一直存在
全局变量	静态	仅在本源文件内	运行时一直存在

函数参数的作用范围和生存期都可以等价于在函数一开始就定义的自动变量。

全局变量和静态局部变量在程序开始运行时就要初始化，因此它们的初值必须是一个编译时就能确定的常数，如：

```
static int m = 2 * 3 + 4; /* 合法，常数表达式的值在编译时即可确定 */
static int n = m; /* 错误，即使 n 定义在 m 的后面也不能用 m 初始化 n */
```

而自动变量是在它产生的时候才初始化的，因此没有这个限制，可以用任何合法的表达式进行初始化。

2.9 指针

指针类型在 C 语言中扮演着重要的角色，因此把它拿出来单独作为一节。指针本身并不是一个单独的数据类型，它总是与其他数据类型联系起来使用。

2.9.1 指针与变量

2.9.1.1 变量的地址

在程序运行时，每个有效的变量都需要占据一定的内存空间。内存的最小单元是字节，每个单元都有独一无二的编号称为地址。CPU 通过给出内存地址来访问相应的内存单元。变量的地址就是它所占据的内存空间中第一个单元的地址。例如，定义如下 3 个变量：

```
long a = 0x11223344; /* long 型变量占据 4 个字节 */
short b = 0x1122; /* short 型变量占据 2 个字节 */
char c = 0x11; /* char 型变量占据 1 个字节 */
```

它们在内存中的一种可能分布如图 2.3 所示。这时变量 a 的地址是 0xBF8B7020，变量 b 的地址是 0xBF8B7024，变量 c 的地址是 0xBF8B7026。

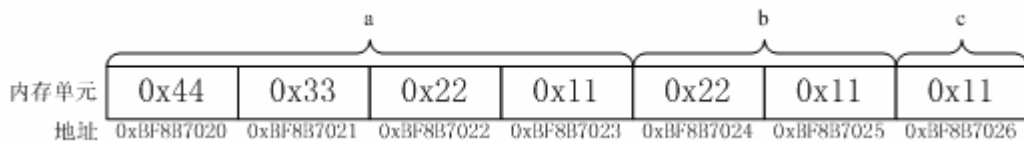


图 2.3 变量的地址

2.9.1.2 变量的指针

简单地说，指针就是用来保存地址的变量，它可以用如下的语法定义：

```
int *p; /* 定义指向整型变量的指针 p, p 的类型是 (int *) */
```

指针的类型总是与它所指向的变量类型相关，任何变量类型都有对应的指针类型。很多时候，“指针”一词也用来代表这种数据类型，就像“整数”一词既可以指一个变量也可以指整型这种数据类型。

定义指针时需要注意的一点是 * 号的使用，如：

```
int *p, q; /* 定义整型指针 p, 整型变量 q */
```

如果这两个变量都要定义成指针，则应该这样写：

```
int *p, *q; /* 定义整型指针 p 和 q */
```

也就是说，所定义的每一个指针前都必须使用一个 * 符号。如果使用 typedef 给指针类型定义一个别名，如：

```
typedef int *int_p; /* int_p 成为 (int *) 型的别名 */
```

之后就可以这样来定义变量：

```
int_p p, q; /* p, q 都是 int_p 型，也就是 (int *) 型 */
```

指针作为一个变量，它可以保存任何值，通常都会让它保存一个有效变量的地址。指针保存一个变量的地址也常说成“指向”了这个变量，例如：

```
int num1, num2; /* 定义整型变量 num1 和 num2 */
int *p = &num1; /* 指针 p 指向了 num1 变量 */
p = &num2; /* 修改指针 p 的值，指针 p 又指向了 num2 变量 */
```

其中 & 操作符用来取得一个变量的地址，它的操作数必须是左值。

变量的地址不能作为左值使用。实际上，全局变量和静态局部变量的地址就是一个常数，因此下面的初始化是合法的：

```
static int num; /* num 是静态变量 */
static int *p = &num; /* num 的地址是常数，可以给静态变量初始化 */
```

但自动变量的地址不是常数，因此不能将它赋给全局变量或静态局部变量，如：

```
void fun(void)
{
```

```
int num;
static int *p = &num; /* 错误，静态变量必须用常数初始化 */
}
```

知道了变量的地址，就可以找到这个变量，这也正是指针的用途。由地址得到变量要用到 * 操作符，如：

```
int num = 1; /* num 的初值为 1 */
int *p = &num; /* 指针 p 指向了变量 num */
(*p)++; /* 通过指针 p 可以访问变量 num */
```



后缀自增自减操作符的优先级高于 * 操作符，不要忘记使用括号。

需要明确的是：* 操作符可以由地址得到“变量”，而不仅是得到“变量的值”，得到的变量既可以作为右值使用，也可以作为左值使用，这是为数不多的表达式的结果可以作为左值使用的例子之一。

* 操作符与 & 操作符的优先级与结合顺序均与逻辑否 ! 相同。

2.9.1.3 指针的长度

在 32 位机器上，内存地址长度都是 4 个字节，与一个长整型数据长度相同，因此不管是什么指针类型，对它使用 sizeof 操作符的结果都是 4。

2.9.1.4 指针的指针

指针作为一种特殊的变量，当然也有自己的地址，它的地址也可以被另外一个指针保存，如：

```
int num = 1; /* 定义变量 num, num 为 (int) 型 */
int *p = &num; /* 定义指针 p, p 为 (int *) 型 */
int **pp = &p; /* 定义指针 pp, 指向 (int *) 型变量的指针为 (int **) 型 */
(**pp)++; /* 通过指针 pp 可以访问变量 p, 进而访问 num */
```

2.9.1.5 指针与 const

指针当然也可以是只读变量，如：

```
int *const p = &num; /* p 是只读的整型指针 */
```

这样定义的指针 p 是不可以作为左值使用的，也就是说它的值不能被修改，因此必须在定义时对它进行初始化，否则它就永远是无效指针。

p 虽然是只读的，但只要 p 指向了有效的变量，*p 是可以访问的。如果改变 const 关键字的位置，如：

```
int const *p; /* p 是指向只读整型变量的指针 */
const int *p; /* const int 与 int const 相同 */
```

这样 p 的类型就完全变了，p 本身不是只读的，可以被修改；但 *p 却是只读的，不能被修改。当然，也可以定义指向只读类型的只读指针，如：



```
int const *const p = &num; /* p 本身是只读的, *p 也是只读的 */
```

C 语言中将只读变量的地址赋给一个指向非只读类型的指针是可以的, 如:

```
int const num = 1; /* 只读变量 num 被初始化为 1 */
int *p = &num; /* 警告, 类型不匹配 */
num = 2; /* 错误, 只读变量不能被修改 */
*p = 2; /* 通过 p 只读变量 num 被修改为 2 */
```

这样, 通过变量的地址就可以“偷偷地”篡改只读变量的值。编译器对这种情况会给出警告, 因此正确使用 `const` 关键字, 有助于发现类似的隐蔽错误。

2.9.1.6 指针类型转换

各种不同的指针类型都可以相互转换, 指针类型转换时, 它保存的地址值不变, 举例如下:

```
int num = 0; /* num 定义为整型变量 */
char *p = &num; /* 警告, 指针类型不兼容 */
p = (char *)&num; /* 强制转换, 消除了上述警告 */
*p = 0xFF; /* 修改 num 的第一个字节 */
*(char *)&num = 0xFF; /* 更直接的写法, 修改 num 的第一个字节 */
```

当指针类型自动转换时, 编译器会给出警告, 如果确实要转换, 可以明确地写出类型转换操作符以消除警告。

实际上对 CPU 来说, 内存地址是无所谓类型的, 内存中的数据也没有类型的概念。指针之所以被分为各种类型, 就是因为它所指向的内存区域可以代表各种不同类型的数据。指针的类型被转换, 也就是它指向的内存区域所代表的数据类型发生了变化。

在 32 位机器上, 指针的长度与整型数的长度相同, 因此两者之间可以无数据损失地相互转换。指针有一种特殊的类型, 它不指向任何具体的数据类型, 可定义如下:

```
void *p; /* p 是 (void *) 型指针 */
```

(void *) 型指针的特点是它与任何其他类型指针自动转换都没有警告, 因此常用来传递任意类型数据的地址。但是对 (void *) 指针使用 * 操作符是无意义的, 因为得到的是无法使用的 void 类型。

2.9.2 指针与操作符

对于指针类型来说, 能进行的操作是非常有限的。常用的操作包括加减法、比较以及逻辑判断。

一、指针与算术操作符

指针可以与一个整数进行加减操作, 结果仍然是同类型的指针。当然, 指针也可以进行自增自减操作和加减法的复合赋值操作, 举例如下:

```
int a[3]; /* 定义有 3 个元素的整型数组 */
int *p = &a[0]; /* 整型指针初始化为数组 a 的第 0 个元素的地址 */
p = p+1; /* p 的值增加了 1*sizeof(int), 指向 a 的第 1 个元素 */
p++; /* p 的值又增加 1*sizeof(int), 指向 a 的第 2 个元素 */
```



```
p += 1; /* 危险, p 的值再增加 1*sizeof(int), 指向未分配的内存 */
```

指针每次加 1, 它保存的地址值就加它所指向的数据类型的长度, 实际上是指向了原来所指变量后面的一个变量。指针加减任何整数, 它的值实际加减的数量都是这个整数乘以它所指向的数据类型的长度, 这是理解指针运算的关键。

两个同类型指针可以进行相减操作, 结果是一个整数, 这个整数是两者保存的地址值之差除以所指的数据类型的长度。这个特性可用来计算两个指针所指位置之间的变量的个数, 如:

```
int n = &a[3] - &a[0]; /* &a[3] 和 &a[0] 这两个指针相减, 得到 3 */
```

指针加减操作的示意如图 2.4 所示。

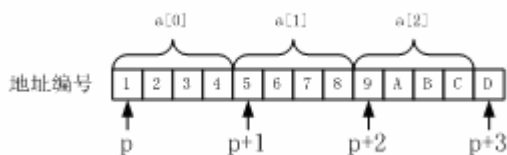


图 2.4 指针加减操作示意



(void *) 型指针指向的是没有具体长度的 void 型数据, 因此不能进行加减操作。

二、指针与比较操作符

指针可以进行比较操作, 比较的结果与比较两个整型数相同。

三、指针与逻辑操作符

指针可以进行逻辑操作, 这时它的值如果为 0 则代表逻辑上的“假”, 值如果不是 0 则代表逻辑上的“真”。合法的变量地址值永远不可能是 0, 所以常用为 0 的指针值表示指针还没有指向任何合法的内存。为了明确, 为 0 的指针值又常被定义为 NULL, 例如:

```
int *p = NULL;
if (p) *p = 0; /* 等价于 if (p != NULL) *p = 0; */
```

注意 NULL 并不是 C 语言关键字, NULL 的定义是 ((void *)0), 由于是 (void *) 型, 它可以与其他任何类型的指针直接进行赋值、比较等操作而不会有警告。

2.9.3 指针与数组

数组类型可以自动转换为一个相应的只读指针类型, 它的值就是它的第 0 个元素的地址, 或者称为首地址, 举例如下:

```
int a[3];
int *p = a; /* a 等价于 &a[0], 由 (int [3]) 型自动转换为 (int *const) 型 */
```

事实上, [] 操作符也要求它的一个操作数为指针类型, a[i] 完全等价于 *(a+i), 为了说明这一点, 可参看下面的代码:


```
l[a] = 0[a]; /* 可以将下标放在 [] 前面, 数组名放在 [] 内, 不提倡这样写 */
p[2] = p[0]; /* 指针 p 获得 a 的首地址后, 就可以通过它访问数组的成员 */
```

因为数组是左值, 所以可以用 & 操作符取地址, 如:

```
int a[3];
int (*p)[3] = &a; /* p 是 (int (*)(3)) 型, &a 是 (int (*const)(3)) 型 */
p++; /* p 保存的地址值增加 sizeof(int [3]) */
```



很多时候, 指向数组第 0 个元素的指针也称为指向数组的指针。

对数组取地址, 得到的是指向这种数组类型的指针。定义指向数组的指针时要注意, 因为 [] 操作符的优先级高于 * 操作符, 所以 * 与变量名要用括号包围起来, 这个括号即使在类型名里也不能去掉。具体分析可参考以下代码:

```
int (*p1)[3]; /* p1 是 (int (*)(3)) 型指向 3 个元素的整型数组的指针 */
int *p2[3]; /* p2 是 (int *(3)) 型包含 3 个整型指针的数组 */
```

再举一个二维数组的例子如下:

```
int a[3][2];
int (*p1)[3][2] = &a;
int (*a1[3])[2] = {a, a+1, a+2};
int *a2[3][2] = {{a[0], a[0]+1}, {a[1], a[1]+1}, {a[2], a[2]+1}};
int (**p2)[2] = a1;
int *(*p3)[2] = a2;
```

这些定义和初始化都是合法的, 具体的分析如表 2.12 所示。

表 2.12 二维数组及指针的类型分析

定义	变量名	类型	数组对应的指针类型	取地址后的类型
int a[3][2];	a	int [3][2]	int (*const)[2]	int (*const)[3][2]
int (*a1[3])[2];	a1	int (*(3))[2]	int (**const)[2]	int (*(**const)[3])[2]
int *a2[3][2];	a2	int *[3][2]	int *(*const)[2]	int *(*const)[3][2]
int (*p1)[3][2];	p1	int (*)(3)[2]	本身就是指针	int (**const)[3][2]
int (**p2)[2];	p2	int (**)[2]	本身就是指针	int (***const)[2]
int *(*p3)[2];	p3	int *(*)[2]	本身就是指针	int *(*const)[2]

总结一下, 可以得出以下规律。

- ◆ 定义变量的语句, 去掉变量名就是它的类型。
- ◆ 将数组类型中的第一个 [] 替换成 (*const) 就得到它对应的指针类型。
- ◆ 对变量取地址, 结果的类型是将定义语句中的变量名替换成 (*const)。

2.9.4 字符串

字符串在 C 语言中并不是单独的数据类型，实际上是以字符数组的形式来处理的。字符串的语法使用双引号作为标记，如：

```
"hello\n"
```

字符串中可以使用空格及各种转义字符。如果两个字符串挨在一起，中间只有空格等空白字符，则被视为一个字符串，如：

```
"hello" "world" /* 等价于 "helloworld" */
```

字符串可用于字符数组的初始化，如：

```
char str[] = "hello";
```

等价于以下初始化方式：

```
char str[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

注意编译器在展开字符串时会自动在最后加一个 ASCII 码为 0 的字符，用于表示字符串的结束，因此 str 的长度被确定为 6 而不是 5。

更多的时候，字符串本身就作为一个字符数组在使用，如：

```
char *p = "hello";  
p[0] = '\0'; /* 编译可通过，运行时产生“段错误”而崩溃 */
```

这里 hello 作为 (const char [6]) 型的数组在使用，数组的初始内容就是 hello。这种数组没有名称，不需要事先定义。编译器碰到这样的字符串，就会自动为它分配空间以保存数据。需要注意的是，在 Linux 操作系统上，编译器将把这种字符串的空间分配在只读内存区上，因此如果试图对它的内容进行修改，程序运行时将引发一个“段错误”而导致崩溃。



由于字符串实质上是字符数组，字符数组类型又可以自动转换为字符指针类型，因此经常直接把字符指针类型就称为字符串类型。

比较以下两种使用字符串的方式：

```
char str1[] = "hello"; /* str1 是字符数组 */  
char *str2 = "hello"; /* str2 是字符指针 */
```

虽然 str1 和 str2 将来都可以作为字符串使用，内容也是相同的，但两者还是有很大区别的。str1 是字符数组，数组的初始内容就是 hello，str1 可代表数组的首地址，但它本身不能被修改，str1 的每个元素却可以被修改；str2 是字符指针，指向一个内容为 hello 的字符数组，str2 本身的值可以被修改，但通过它修改所指向数组的元素则会引发一个运行时错误。

有些编译器能够对放在只读内存区的字符串进行优化合并，举例如下：

```
char *s1 = "hello world";  
char *s2 = "hello world";  
s2-s1; /* 优化后，s2 与 s1 指向相同的存储空间，结果为 0 */
```



2.9.5 指针与结构体

指针可以指向结构体变量，如：

```
struct my_struct {
    int m, n;
} var = {1, 2}; /* 定义结构体类型 my_struct 和结构体变量 var */
struct my_struct *p = &var; /* p 是指向结构体的指针 */
(*p).m = (*p).n; /* 用 * 操作符由指针得到变量，再用 . 操作符得到成员 */
```

鉴于从指针得到所指向结构体的成员这一操作很常用，因此 C 语言中引入了一个专门的操作符，用法举例如下：

```
p->m = p->n; /* p->m 等价于 (*p).m */
```

-> 操作符与 [] 操作符的优先级和结合顺序相同。

在实现链表数据结构时，不可避免地要用到类似下面的结构体定义：

```
struct list_head {
    int data; /* 链表节点中的数据 */
    struct list_head *next; /* 指向链表中的下一个节点 */
    /* 这时 struct list_head 还没有完整定义 */
}; /* 到这里，struct list_head 才定义完整 */
```

从这里可以得到一点启发，即 C 语言中允许定义指向一个未完整定义的结构体的指针，参见以下的示例代码：

```
void fun(void)
{
    struct my_type *p = NULL; /* 合法，虽然 struct my_type 还没有定义 */
    p->m = 0; /* 非法，访问结构体变量时其类型必须有完整定义 */
}

struct my_type { int m; };
```

如果想进行类似于以下代码的相互嵌套的结构体定义：

```
struct A {
    struct B b; /* 错误，struct B 还没有完整定义 */
};

struct B {
    struct A a;
};
```

这是不可能实现的，结构体中结构体成员的类型必须有完整定义，因此无论哪一个结构体定义在前，总会有类型未完整定义的错误，但可以实现以下的定义：

```
struct B; /* 有些编译器要求先声明 B 为结构体，称为前向声明 */
```



```
struct A {
    struct B *pb;
};

struct B {
    struct A *pa;
};
```

2.9.6 指针与函数

一般来说，一个函数将被编译为一段可执行代码，运行时这段代码放在内存中，有固定的起始地址，称为入口地址。函数调用的实质就是利用跳转指令让 CPU 的执行流程转移到函数的入口地址处。

2.9.6.1 函数指针

对函数类型使用取地址操作符 &，就可以得到它的入口地址，有相应的函数指针类型可用来定义一个指针以存放入口地址，如：

```
int sum(int a, int b); /* 函数声明，sum 是 (int (int, int)) 型函数 */
int (*pfun)(int, int) = &sum; /* pfun 的类型是 (int (*)(int, int)) */
```

因为 * 操作符的优先级低于表示函数的小括号，因此定义函数指针时必须用小括号将它和变量名包围起来。如果定义成这样：

```
int *pfun(int, int); /* 返回值为 (int *) 型指针的函数 */
```

就成为了声明一个返回整型指针的函数。

事实上，对函数使用 & 操作符是不必要的，因为函数类型可以自动转换为函数指针类型，它的值就是函数的入口地址，如：

```
int (*pfun)(int, int) = sum; /* sum 等价于 &sum */
```

函数调用本质上是通过函数指针来实现的，如：

```
int result;
result = (&sum)(2, 3); /* 通过入口地址调用函数 sum */
result = pfun(2, 3); /* pfun 的值是 &sum，可以用来调用函数 sum */
result = sum(2, 3); /* sum 转换为 &sum，可以用来调用函数 sum */
```

这样，函数名、函数的入口地址、函数指针三者就达到了内在的一致。从这里可以发现，对函数指针使用 * 操作符得到的就是它自己所保存的值。进一步可得如下结论。

- ◆ &sum 是函数 sum 的入口地址。
- ◆ sum 自动转换为函数指针类型，值为 &sum。
- ◆ *sum 等价于 *&sum，即 sum，然后 sum 又自动转换为函数指针类型，值为 &sum。

如此可以连续对函数指针应用多次 * 操作符，其值不变。这是 C 语言中一个有趣的现象。



2.9.6.2 指针参数

函数的参数可以是指针类型，看下面的例子：

```
void swap(int *a, int *b) /* 用于交换两个整型变量的值 */
{
    int tmp; /* 临时变量，暂存一个整型的值 */
    tmp = *a; *a = *b; *b = tmp; /* 通过传入的指针访问调用者的变量 */
}
```

调用 `swap` 函数时，传入变量的地址：

```
int a = 1, b = 2;
swap(&a, &b); /* 将变量的地址传给 swap 函数 */
```

这样，两个变量的值就真正被交换了。C 语言的函数实际上只能“按值传递”参数，但通过指针参数，也可以间接地实现类似其他高级语言的“按地址传递”参数。

指针参数还经常用来让函数“返回”除了它的返回值之外的其他数据，举例如下：

```
int get_value(int *v)
{
    *v = 1000; /* 设置参数指向的变量的值 */
    return 0; /* 函数返回值 */
}
```

调用 `get_value` 函数时，传入变量的地址，调用完成后即得到数据：

```
int value;
get_value(&value); /* 函数返回值被忽略，value 得到数值 1000 */
```

这种为了让调用者得到数据的参数有时称为“输出”型的参数。当函数需要得到的数据量较大时，比如是数组或结构体，通常也会采用指针参数的方式。为了避免函数内通过指针修改调用者提供的数据，可使用 `const` 关键字，如：

```
char get_char_at(const char *str, int index)
{
    return str[index]; /* 返回得到的字符 */
}
```

这个函数的作用是得到字符串 `str` 的第 `index` 个字符。调用方法如下：

```
char ch = get_char_at("ABCD", 2); /* ch 的值是 'C' */
```

这种参数有时称为“输入”型的参数，当然，通过指针参数也可以传递既是“输出”型又是“输入”型的参数。

如果将上述函数的声明写成如下形式：

```
char get_char_at(const char str[], int index);
```

则与原来的定义完全等价。函数的参数不能为数组，因此编译器自动将 `[]` 理解为指针型。



使用【】唯一可能的好处是提高可读性，让代码读者知道这个参数需要的是数组，并且能指明数组的大小，如：

```
void need_array(int a[8]); /* 提供了 a 应该是指向具有 8 个元素的数组这个信息 */
```

实际上完全等价于：

```
void need_array(int *a); /* 看到这个原型时，并不知道 a 应该指向什么 */
```

为了在函数内能够知道数组元素的确切个数，通常会传递两个参数，一个参数是数组的首地址，另一个参数是数组中元素的个数，如：

```
void set_zero(char *buf, int len) /* 将元素个数为 len 的数组 buf 清零 */
{
    char *p;
    for (p = buf; p < buf+len; p++) *p = '\0';
}
```

2.9.6.3 数组指针参数

虽然函数参数类型上的【】将被理解为指针型，但也不能一概而论，看下面的例子：

```
int sum(int a[][3])
{
    return a[0][0]+a[0][1]+a[0][2]; /* 返回数组中三个元素的和 */
}
```

调用方法如下：

```
int s;
int a[3] = {1, 2, 3};
s = sum(&a); /* 得到 a 的三个元素的和，即 6 */
```

也就是说，在函数参数上，只有第一个【】被理解为指针，其后的【】表明指针指向数组类型，此数组中的元素个数必须明确给出。

2.9.7 回调函数

在调用函数时将另一个函数的指针作为参数传递进去，这样在被调用的函数里就可以通过指针间接地调用这个函数。程序的执行流程从调用者进入被调用者，然后又进入由调用者指定的一个函数，因此称这个函数为回调函数。举例如下：

```
int add(int a, int b) /* 此函数返回两数之和 */
{
    return a+b;
}

int sub(int a, int b) /* 此函数返回两数之差 */
{
    return a-b;
}
```



```

}

int call_fun(int (*fun)(int, int), int a, int b)
{
    return fun(a, b);
}

```

调用函数 `call_fun`:

```

int result;
result = call_fun(add, 3, 2); /* 最终调用了 add(3, 2) */
result = call_fun(sub, 3, 2); /* 最终调用了 sub(3, 2) */

```

这样实际上使得调用者可以指定被调用函数的功能。

有时需要让回调函数知道最初的调用者提供的有关数据,这时可以使用一种称为“上下文”的参数。比如要将上面的例子进行改动,不通过返回值得到结果,而是通过指针参数得到结果,则可以使用这一技术,代码修改如下:

```

void add(void *context, int a, int b) /* 此函数返回两数之和 */
{
    int *result = context;
    *result = a+b;
}

void sub(void *context, int a, int b) /* 此函数返回两数之差 */
{
    int *result = context;
    *result = a-b;
}

void call_fun(void (*fun)(void *, int, int), void *context, int a, int b)
{
    /* 在此函数中对 context 不做任何处理,直接传递给回调函数 */
    fun(context, a, b);
}

```

调用函数 `call_fun`:

```

int result;
call_fun(add, &result, 3, 2); /* 最终调用了 add(&result, 3, 2) */
call_fun(sub, &result, 3, 2); /* 最终调用了 sub(&result, 3, 2) */

```

使用“上下文”参数时,被调用的函数根本不关心参数所指向的内容,而是直接将其传递给回调函数。“上下文”参数采用 `(void *)` 型的好处是可以与其他任意类型指针自动转换,此时,调用者和回调函数必须将其视为相同的数据类型。

2.9.8 函数指针类型转换

函数指针类型可以与其他指针类型相互强制转换。例如,先定义一个函数:



```
int add(int a, int b)
{
    return a+b;
}
```

可以这样调用：

```
void *p = add; /* 函数指针转换为 (void *) 型指针 */
((int (*)(int, int))p)(2, 3); /* 将 p 转换成函数指针以后调用函数 */
```

在极个别情况下会把一种函数指针类型转换成另外一种函数指针类型，然后进行函数调用。仍然利用上述的 `add` 函数来说明，可以这样调用：

```
int (*pfun2)(int, int) = add; /* pfun2 是有两个 int 型参数的函数指针 */
/* 将 pfun2 强制转换为有三个 (int) 型参数的函数指针 */
int (*pfun3)(int, int, int) = (int (*)(int, int, int))pfun2;
pfun3(2, 3, 4); /* 通过 pfun3 调用函数，得到 5 */
```

这样调用是能够通过，一般情况下，执行流程进入到被调用函数后，多余的参数将被忽略，能够得到正确的结果。但如果这样调用：

```
int (*pfun2)(int, int) = add; /* pfun2 是有两个 int 型参数的函数指针 */
/* 将 pfun2 强制转换为有一个 (int) 型参数的函数指针 */
int (*pfun1)(int) = (int (*)(int))pfun2;
pfun1(2); /* 通过 pfun1 调用函数，结果不确定 */
```

因为函数调用时，调用者和被调用者对参数存放的位置有固定的约定，如果调用时少提供了一个参数，则被调用者仍然试图从约定的位置取得数据，结果将是不确定的，甚至可能导致程序运行崩溃。

2.10 预处理语句

C 语言程序的整个编译过程可分为四个阶段：第一阶段为预处理，代码中的所有预处理语句得到处理，经过预处理后，代码中就不再包含任何预处理语句；第二阶段为编译，把 C 语言的代码转化成与目标机器平台相关的汇编代码；第三阶段为汇编，把汇编代码进一步转化为机器指令，称为目标代码；第四阶段为链接，将一个或多个由源代码生成的目标代码文件与系统提供的代码库相互链接形成可执行程序。

预处理语句与其他 C 语言代码的语法完全不同，它以行为单位，到一行结束时语句就结束，并且都以符号 `#` 开头。如果预处理语句太长需要折行时，可以在行尾加一个反斜杠 `\` 表示续行。

2.10.1 文件包含

使用 `#include` 语句可以实现文件包含，例如：

```
#include <stdio.h> /* 全局包含 */
#include "my_def.h" /* 局部包含 */
```

预处理时，`#include` 语句将被替换为语句中所指定的文件的内容。如果这个文件中又包含预

处理语句，则也要进行处理，直至内容中不包含任何预处理语句为止。

#include 语句可以用来包含任何类型的文件。一般情况下被包含的文件名会以 **.h** 作为后缀，称为头文件。头文件通常作为编程接口使用，内容一般是定义的常数、类型、函数的声明等。

文件包含有全局包含和局部包含两种方式，它们的区别如下。

- ◆ 全局包含使用尖括号 **<>** 包围文件名，使用这种方式时，编译器将在指定的系统头文件目录中搜索所需的文件。调用系统函数时所需的头文件一般用这种方式包含。
- ◆ 局部包含使用双引号 **"** 包围文件名，使用这种方式时，编译器在源码文件本身所在的目录中搜索相应的文件，如果找不到，再到指定的系统头文件目录中搜索。使用程序自己的头文件时一般用这种方式。

需要注意的是，这种搜索并不进入子目录，因此必要时可使用文件的相对路径或绝对路径。

2.10.2 宏定义

2.10.2.1 无参数宏的定义

定义无参数宏的语法举例如下：

```
#define PI 3.1415926 /* 定义了一个宏，名称为 PI */
```

#define 后面的标识符是宏的名称，其后的所有内容（包括空格，但不包括注释）是宏所代表的字符串，也称为这个宏的值。定义宏的作用是：当其后的代码中出现了宏名时，将被替换为宏所代表的整个字符串，这种替换是在预处理阶段进行的，称为宏的展开。例如，在上面的宏定义后如果出现以下代码：

```
float s = 2*PI*r; /* 代码中包含宏 PI */
```

实际上将展开成：

```
float s = 2*3.1415926*r;
```

但是对于 C 语言字符串中的宏名并不展开，如：

```
char *str = "macro is PI"; /* 字符串中的宏名保持原样 */
```

编程时常将一些难记的、没有明确意义的常数定义为名称好记、有意义的宏，以使程序利于阅读。另外，将代码中经常修改的常数定义为宏有助于提高程序的可维护性。通常，宏的名称都采用全大写的方



与 C 语言的其他语句不同，定义宏的语句末尾不需要分号。

宏的值也可以定义成空的。定义宏时如果宏的值中有其他宏出现，则也要展开，例如：

```
#define PI2 PI*2 /* 将展开成 #define PI2 3.1415926*2 */
```

使用宏的时候需要注意，宏的展开仅仅是简单的字符串替换，不考虑任何语法上、逻辑上的合理性。例如，先定义一个宏：

```
#define SUM 1+2 /* 定义 SUM 表示 1+2 */
```

在表达式中使用这个宏：

```
SUM*3; /* 感觉上像是 (1+2)*3 */
```

实际上将展开成：

```
1+2*3;
```

如果定义宏时在它的值上加上小括号，就可避免这一问题，如：

```
#define SUM (1+2)
```

定义过的宏还可以用 `#undef` 语句取消定义，如：

```
#undef PI /* 取消 PI 的宏定义 */
```

其后的代码中如果再出现 `PI` 就不会被展开了。

2.10.2.2 带参数宏的定义

宏也可以带参数，这使得宏的用途更广泛，例如：

```
#define SQUARE(i) ((i)*(i)) /* i 是参数 */
```

使用带参数宏时要提供参数，参数放在宏名后的小括号内，仅代表一个字符串，可以包含空格等各种字符，如：

```
sqr = SQUARE(a+b); /* 所提供的参数是 a+b */
```

带参数宏展开时，所提供的参数将替换展开后的字符串中相应的参数名。比如上述例子将展开为：

```
sqr = ((a+b)*(a+b));
```

注意小括号的使用，如果定义宏时不在每个参数名上加小括号，如：

```
#define SQUARE(i) (i*i)
```

那么同样的例子将展开为：

```
sqr = (a+b*a+b);
```

显然跟预想的意义是不一致的。

如果希望参数出现在字符串中，可使用如下的宏定义：

```
#define str(s) "s" /* 试图使参数 s 出现在字符串中 */
```

这样字符串中的内容永远保持原样，因此不论参数 `s` 给什么内容，展开后得到的都是字符串

s。必须使用下面的语法才能实现将参数放在字符串中：

```
#define str(s) #s /* 使用符号 # 表示参数要变成字符串 */
```

这样定义后，`str(hello)` 将展开成 `"hello"`。

有时候需要参数作为展开的代码中某个名称的一部分出现，则必须用两个 `#` 符号将参数与名称的其他部分隔开，如：

```
#define get(v) get_##v(void)
```

这样定义后，`get(name)` 将展开成 `get_name(void)`。

2.10.3 宏与函数

带参数宏的定义和使用形式均与函数类似，实际上可以用它来模拟函数，如：

```
#define swap_int(a, b) \
do { \
    int tmp; \
    tmp = (a); (a) = (b); (b) = tmp; \
} while (0)
```

其中 `do while(0)` 的作用是为了让它更像一个函数，这样在使用宏的时候末尾就必须加上分号才是完整的语句。使用方法如下：

```
int a = 1, b = 2;
swap_int(a, b); /* 如果定义时没有 do while(0)，末尾的分号可以不加 */
```

实际上将展开为：

```
do {int tmp; tmp = (a); (a) = (b); (b) = tmp } while (0);
```



不要担心 `do while(0)` 会生成没有实际意义的目标代码，现代的编译器完全可以将其优化掉。

上面定义的宏因为内部用了一个 `int` 型临时变量，故不适用于其他类型变量。可以使用 `gcc` 扩展的操作符 `typeof` 使之更具通用性，如：

```
#define swap(a, b) \
do { \
    typeof(a) tmp; \
    tmp = a; a = b; b = tmp; \
} while (0)
```

`typeof` 操作符可以得到一个表达式的类型，所得到的类型可以再用来定义变量。

如果要模拟有返回值的函数，可以采用在复合语句外面加上小括号变成表达式的办法。例如，可以用以下的宏求字符串的长度：

```
#define str_len(s) \
({ \
```

```

char *p; \
char *tmp = (s); \
for (p = tmp; *p != '\0'; p++); \
p=tmp; \
})

```

在很多情况下，使用宏与函数能达到相同的目的，但它们的区别还是很明显的。

- ◆ 宏在预处理阶段处理，函数在编译阶段处理。
- ◆ 宏展开时不检查参数类型，函数调用时要检查形参与实参的类型是否匹配。
- ◆ 使用宏没有调用开销，使用函数时要生成函数调用的代码，有开销。
- ◆ 宏可以直接修改传给它的变量，函数内要想修改调用者的变量则需要传递变量地址。
- ◆ 多次使用一个宏，等于宏所代表的代码重复写多次，多次调用一个函数则只生成函数调用的代码，函数本身的代码只有一份。



如果编译器支持，可以使用内联函数代替宏模拟的函数。

2.10.4 代码分支

预处理可以根据指定的条件，保留或去掉源代码中的一些行，如：

```

#if DEF == 1
code1;
#elif DEF == 2
code2;
#else
code3;
#endif

```

以上代码的功能是：当 DEF 定义为 1 时，code1 部分被保留，code2 和 code3 部分被去掉；当 DEF 定义为 2 时，code2 部分被保留，code1 和 code3 部分被去掉；当 DEF 定义为其他值时，code3 部分被保留，code1 和 code2 部分被去掉。



#elif 很容易误写为 #else if。

#if 和 #elif 所判断的条件表达式与 C 语言本身的表达式很类似，但需要注意，条件中只能出现常数，不能出现 C 代码中的变量。

另外两种判断条件的方式是 #ifdef 和 #ifndef，#ifdef 用于判断一个宏是否被定义过，相反，#ifndef 用于判断一个宏是否未被定义过，例如：

```

#ifdef DEBUG
printf("debug string\n"); /* 如果 DEBUG 被定义过，则语句生效，否则不生效 */
#endif

```

#if 0 常用来屏蔽暂时不用的代码，如：



```
#if 0
int sum(int a, int b)
{
    return a+b; /* 返回两数之和 */
}
#endif
```

因为 `/* */` 注释不能嵌套，所以必须采用这种方式。代码分支是可以嵌套的。

在较复杂的程序里，可能会出现一个头文件中包含另一个头文件的情形，如果有一个源文件又同时包含了两者，则等于其中一个头文件被包含了两次，于是会出现大量的重复定义。为了防止出现这种重复包含现象，可在头文件中加入如下代码：

```
#ifndef __NAME_H
#define __NAME_H
/* 不可重复的内容放在这里 */
#endif
```

这里采用的技巧是用宏 `__NAME_H` 有没有定义来判断这个头文件的内容有没有被包含过。如果 `__NAME_H` 没有定义过，则说明头文件的内容还没有被包含过，那么就定义这个宏，同时头文件的内容也是生效的。如果 `__NAME_H` 已定义过，说明头文件的内容已被包含过，则从 `#ifndef` 到 `#endif` 之间的内容全部被去掉，因此不会有重复的内容产生。



第 3 章 开发环境

本章的内容可以分为以下三大部分。

一、Linux 的使用

这一部分包括 **Linux 使用基础**、**Linux 常用命令**、**Shell 使用进阶**三个小节，目的是让初次接触 **Linux** 操作系统的读者尽快熟悉进行嵌入式开发所需的基本操作，以及 **Linux** 操作系统上的一些基本概念。

二、Debian 5.0 操作系统及开发环境的安装

这一部分包括 **Debian 5.0** 的安装与使用和建立交叉编译环境两节，目的是指导读者从无到有建立嵌入式开发所需的环境。本书其他章的内容均建立在这个开发环境的基础上。

三、开发工具的使用

本章中剩余的几节属于这一部分。这部分的目的是让读者掌握一些基本开发工具的使用，如编辑器、编译器等，以便进行嵌入式系统的开发。本书中的所有例程都是用这里所介绍的工具进行开发的。

在 **Linux** 的命令行上以及 **Makefile** 中，符号 **#** 被认为是注释内容的开始，本章中对所用的命令进行了很多注释，希望读者注意。

3.1 Linux 使用基础

严格来讲，**Linux** 只是一个操作系统内核，它实现了操作系统的核心功能：内存管理、进程管理、文件系统管理和设备管理等。内核提供标准的系统调用接口供应用程序使用。用户要操作计算机，还必须有很多应用程序的支持。**Linux** 内核加上不同的应用程序集合，就形成了风格迥异的各种发行版。

Shell 是一个主要的应用程序，它提供其他程序启动的环境，并且提供一个命令行界面与用户进行交互。**Linux** 上常用的 **Shell** 程序有 **sh**，**bash**，**csh** 等，其中 **sh** 是符合标准的 **Shell**，其他则在 **sh** 的基础上有不同的扩展。

在 **Shell** 的命令行界面中可输入各种命令进行操作。这些命令可分为两类：一类是由 **Shell** 自己实现的功能，称为内置命令；另一类本身是一个单独的应用程序，命令输入完毕后 **Shell** 将执行这个程序。**Linux** 中的命令大多数都属于后者，称为外部命令。

Linux 的各种发行版以及各种不同版本在使用上可能会有一些差异，本章介绍的内容均以 **Debian 5.0 (lenny)** 操作系统为准。

3.1.1 命令参数与选项

大多数 Shell 命令都接受命令参数，参数写在命令后面，用空格隔开。参数可能表示一个文件名，也可能表示一个命令选项。命令选项有两种约定俗成的形式：长格式和短格式。

长格式命令选项用两个减号加英文单词来表示。例如大多数命令都支持 `--help` 选项，用来显示命令的帮助信息：

```
man --help # 显示 man 命令的帮助信息
```

短格式命令选项用一个减号加一个字母来表示。多个短格式选项往往还可以合并在一起使用，如：

```
man -h # 等价于 man --help  
ls -al # 等价于 ls -a -l
```

3.1.2 文件、目录和路径

文件、目录以及路径是操作系统中的基本概念。Linux 系统上的文件概念比较宽泛，它可以是用来保存数据的普通文件，也可以代表一个设备，还可以是用来通信的命名管道等。应用程序通过文件名来访问文件。文件名中可以包含除斜杠 / 之外的任何字符，而以句点 . 开头的文件名就被系统认为是隐藏文件。



尽量不要在文件名中出现 `*`, `?`, `$`, `\`, `&`, `>`, `<` 这些字符，它们本身都有一些特殊含义，虽然系统并不禁止在文件名中使用，但往往会给操作带来一些不必要的麻烦。

目录也可以看做是一个特殊的文件，只不过它记录的是其他文件的信息。同一个目录里不能有名字相同的文件。目录与它记录的文件形成包含关系。如果目录 A 包含目录 B，则 A 称为 B 的父目录，B 称为 A 的子目录。

Linux 的文件系统中只能有一个根目录，它没有父目录，其他目录都必须有父目录。每个目录（包括空目录）下默认都有两个目录，其中一个的名字是一个句点 . 代表目录自身，另一个的名字是两个句点 .. 代表父目录。根目录下的 .. 目录是个例外，因为它没有父目录，所以它也代表根目录自身。由于目录名以句点开头，所以它们都是隐藏文件。实际上，它们都是相应目录的硬链接。

用户在 Shell 中操作时，有一个当前目录的概念。当前目录是很多命令默认的操作目录，并且是相对路径查找的起点。一般来说，用户有一个家目录，放在根目录的 home 目录下，以用户的名字命名。家目录是一个用户拥有完全操作权限的目录。家目录通常可以用波浪线 ~ 这个名字代表。

路径用来表示一个文件，形式上由斜杠 / 隔开的多个文件名组成。除最后一个文件名外，其他的文件名都必须代表目录。每个斜杠后的文件必须存在于斜杠前的目录中，这样形成一个逐级查找的过程。路径有两种表达形式：以斜杠开始的路径称为绝对路径，将从根目录开始逐级查找；不以斜杠开始的路径称为相对路径，将从当前目录开始查找。

3.1.3 用户与权限

Linux 是多用户的操作系统，多个用户可以同时登录进行操作。不同用户有不同的权限。**root** 用户拥有最大的权限，其他用户的权限则有相应的限制。例如 **root** 用户可以使用 **insmod** 命令加载内核模块，而普通用户则不允许。

Linux 系统上还有用户组的概念，一个组可以包含多个用户，一个用户也可以同时加入多个组。组内用户除了自身的权限外，还拥有用户组成员的共同权限。通常每建立一个新的用户，也同时建立一个与用户同名的组，新用户包含在这个组内。

Linux 文件系统每个文件都有权属标志，系统根据这些标志来控制用户对这个文件的访问。首先，每个文件都记录了它所属的用户和所属的组；其次，每个文件都有三组权限标志，分别针对所属用户、所属组和其他用户，每组权限标志又包括读、写、执行三种权限，可以分别设为开或者关。如果用户对一个文件有读权限，那么他就可以查看这个文件的内容；如果有写权限，就可以修改文件的内容；如果有执行权限，就可以把这个文件作为可执行程序来运行。

对于一个目录来说，同样有这三种权限。如果用户对一个目录有读权限，那么他就可以查看目录的内容，也就是目录内包含的文件列表；如果有写权限，就可以修改目录的内容，即建立或删除文件；如果有执行权限，用户就可以使用此目录作为路径操作目录下的文件，否则不允许操作，并且用户不能将此目录作为当前目录。

如果用户既是文件的所属用户，又是所属组的成员，则访问权限由前者的权限标志决定。需要指出的是，**root** 用户作为特权用户，对文件的操作不受权属标志限制。



当用户对一个目录有写权限时，他可以删除其内的任何文件，即使这个文件属于其他用户甚至是 **root**。

文件的权属标志可用命令 **ls -l** 来查看，如图 3.1 所示是对输出结果的解释。

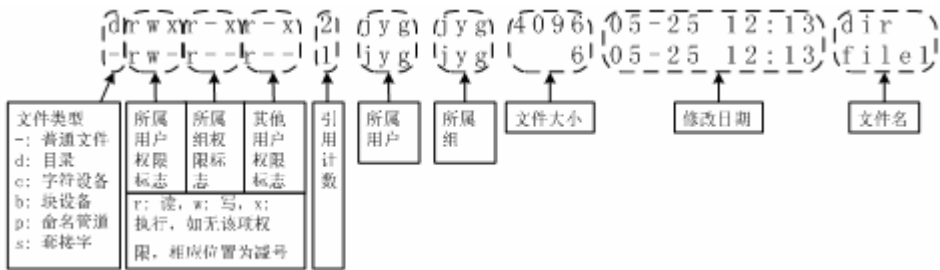


图 3.1 查看文件权限输出结果释义

3.1.4 硬链接与符号链接

Linux 使用的主流文件系统（如 **ext2**, **ext3** 等）都支持硬链接和符号链接。在这些文件系统中，文件与文件的内容并不是固定的一一对应关系。保存文件内容的实体称为索引节点，每个索引节点由它的索引号代表，这些索引号在整个文件系统中是不重复的。而用户看到的文件实质上仅是所在目录中的一条记录，这条记录包含了文件名称和一个索引号，这样就建立了文件名到索引节点的对应关系，这种关系被称为硬链接。

一个文件名只能并且必须硬链接到一个索引节点，但一个索引节点却可以被多个文件链接。也就是说，用户在文件系统中看到的不同文件，实际上可能链接到同一个索引节点，如果是这样，这些文件的内容就会一直保持相同，并且只占同一份磁盘空间。

索引节点有一个重要属性是引用计数，记录着指向它的硬链接的个数。当一个新建的文件硬链接到某个索引节点时，这个节点的引用计数增加 1；当链接到某个索引节点的一个文件被删除时，这个节点的引用计数减去 1，这时要判断引用计数是否已下降到 0，如果是，则这个索引节点就被真正从文件系统中删除。

显然，硬链接只能在同一个文件系统（磁盘）上有效，因为每个文件系统对自己的索引节点都是独立编号的。符号链接克服了这一缺点，实际上它可以看成是一个单独的文件，这个文件的内容是另外一个文件的路径。当用户操作符号链接时，系统根据文件的标志位判断出它是一个符号链接，于是进行特殊处理，将操作转移到它所指向的文件上。事实上，符号链接完全可以指向一个不存在的目标文件。

硬链接与符号链接示意如图 3.2 所示，其中 file1 和 file2 是到同一个索引节点的硬链接，file3 是到 file2 的符号链接，实际上从这三个文件名看到的文件内容都是相同的。

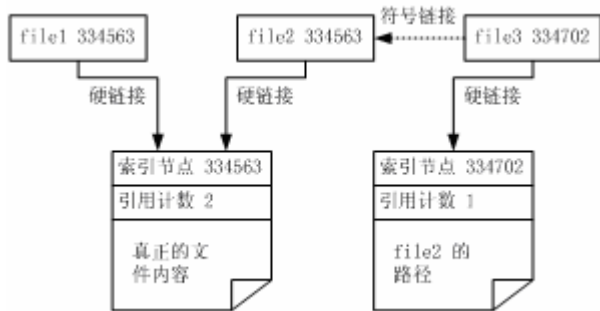


图 3.2 硬链接与符号链接示意

3.1.5 命令使用技巧

Debian 5.0 操作系统采用 `bash` 作为 Shell。在 `bash` 的命令行界面上有许多提高输入效率的办法，下面列举一些。

- ◆ 输入命令未完成时可按 `Tab` 键，Shell 可以自动补齐；如果存在多个可能性，则不能自动补齐，再按一下 `Tab` 键将显示所有可能性。文件名、命令都可以自动补齐。
- ◆ 按上、下键头键可以在命令的执行历史中查找命令。
- ◆ 在命令输入过程中（未按回车键）可按 `Ctrl+C` 组合键取消输入。
- ◆ 正在执行的命令可按 `Ctrl+C` 组合键中止。

3.2 Linux 常用命令

本节将介绍 Linux 操作系统上的常用命令。这些命令的用法在不同的发行版上基本是一致的。在介绍命令的使用时，方括号 `[]` 括起来的内容表示可选项，用竖线 `|` 隔开的几项内容表示可任选其一，省略号 `...` 表示前一项内容可以有多个。

3.2.1 查阅手册

3.2.1.1 man

man 命令用来查阅 Linux 上的手册页，基本用法如下：

```
man [n] item
```

其各个参数的含义解释如下。

- ◆ **n**：节号，一般为从 1 到 8 的数字，1 代表可执行程序，2 代表系统调用，3 代表 C 库函数等。它是可选项，如果省略，将从第 1 节开始查找。
- ◆ **item**：需要查阅的条目。

使用举例：

```
man write # 查阅命令 write 的手册页
man 2 write # 查阅系统调用 write 的手册页
man man # 查阅 man 命令的手册页
```

查阅手册页时，用 **Ctrl+F** 组合键向下翻页，用 **Ctrl+B** 组合键向上翻页；输入字符 **/** 再输入待查找字符串可进行查找，查找时输入字符 **n** 表示查找下一个，输入字符 **N** 表示查找上一个，用 **q** 键退出。



查阅手册页时的基本操作与所用的查阅程序有关，默认为 **pager**。

3.2.1.2 whatis

whatis 命令可用来查询某个条目出现在手册页的哪些节中，基本用法如下：

```
whatis item
```

其参数的含义解释如下。

- ◆ **item**：需要查阅的条目。

3.2.1.3 info

info 命令用于查阅 Info 格式的帮助文档，基本用法如下：

```
info [item]
```

其参数的含义解释如下。

- ◆ **item**：需要查阅的条目。可选项，如果省略，将显示最顶层的 **Info** 文档。

3.2.2 文件相关

3.2.2.1 ls

ls 命令用来列出文件信息，基本用法如下：

```
ls [-a] [-l] [-h] [-d] [-i] [file ...]
```

其各个参数的含义解释如下。

- ◆ -a: 显示所有文件，包括隐藏文件，文件名以 . 开头的就是隐藏文件。
- ◆ -l: 输出文件的详细信息。
- ◆ -h: 当同时使用 -l 时，以人类可读方式显示文件大小（用 KB, MB 等）。
- ◆ -d: 当 file 是目录时，显示目录本身的信息而不是目录内文件的信息。
- ◆ -i: 显示文件的索引号和引用计数。
- ◆ file: 如果是目录，则列出目录中的文件；如果不是目录，则列出该文件；如果省略，则列出当前目录下的文件。

使用举例：

```
ls # 列出当前目录下的文件
ls -al # 列出当前目录下的所有文件的详细信息
ls / # 列出根目录下的文件
```

3.2.2.2 mkdir 和 rmdir

mkdir 用于创建一个新目录，基本用法如下：

```
mkdir [-p] [-v] file ...
```

其各个参数的含义解释如下。

- ◆ -p: 如果 file 的父目录不存在，则先创建父目录。
- ◆ -v: 对于创建的每个目录输出一条说明。
- ◆ file: 要创建的目录。

rmdir 用于删除目录（目录必须为空），基本用法如下：

```
rmdir [-p] [-v] file...
```

其各个参数的含义解释如下。

- ◆ -p: 同时删除父目录，如 rmdir -p a/b/c 等价于 rmdir a/b/c, a/b, a。
- ◆ -v: 对于删除的每个目录输出一条说明。
- ◆ file: 要删除的目录。

3.2.2.3 cd 和 pwd

cd 是内置命令，用于更改当前目录，基本用法如下：

```
cd [file]
```

其参数的含义解释如下。

◆ **file**: 更改到 **file** 目录, 如果省略, 则更改到当前用户的家目录。

pwd 命令用于显示当前目录的绝对路径, 用法非常简单, 不需要参数。

3.2.2.4 cp 和 ln

cp 命令主要用于复制文件, 基本用法如下:

```
cp [-r] [-l] [-s] [-v] [-d] src ... dst
```

其各个参数的含义解释如下。

- ◆ **-r**: 递归复制 (复制目录和目录内的所有文件)。
- ◆ **-l**: 建立硬链接而不复制文件内容。
- ◆ **-s**: 建立符号链接而不复制文件内容。
- ◆ **-v**: 输出说明文字。
- ◆ **-d**: 复制符号链接本身而不是它指向的目标。
- ◆ **src**: 源文件, 如果是多个, **dst** 必须为目录。
- ◆ **dst**: 目标文件, 如果是目录, 则在该目录下建立文件; 如果不是目录, 则建立 (或替换) 该文件。

ln 命令只能用于建立链接, 基本用法如下:

```
ln [-f] [-s] [-v] target [file]
```

其各个参数的含义解释如下。

- ◆ **-f**: 如果 **file** 已存在, 则将其强制删除。
- ◆ **-s**: 建立符号链接而不是硬链接。
- ◆ **-v**: 输出说明文字。
- ◆ **target**: 新链接指向的目标文件 (如为符号链接, 则相对路径相对于 **file**, 不是当前目录)。
- ◆ **file**: 新链接的名称, 如省略, 则在当前目录下建立与 **target** 同名的文件。

3.2.2.5 mv

mv 命令用于移动文件, 基本用法如下:

```
mv [-f] [-v] src ... dst
```

其各个参数的含义解释如下。

- ◆ **-f**: 目标存在时, 直接覆盖, 不提示。
- ◆ **-v**: 输出说明文字。

- ◆ **src**: 源文件, 如果是多个, 则 **dst** 必须为目录。
- ◆ **dst**: 目标文件, 如果是目录, 在该目录内建立文件; 如果是文件, 则建立 (覆盖) 该文件。



源文件与目标文件在同一个目录下时也可视为“移动”, 实际上就是重命名。

使用举例:

```
mv old new # 将当前目录下的 old 重命名为 new
```

如果 **new** 是一个已存在的目录, 则上述命令的作用就是将 **old** 移动到 **new** 内。

3.2.2.6 rm

rm 命令用于删除文件, 基本用法如下:

```
rm [-r] [-f] [-v] file ...
```

其各个参数的含义解释如下。

- ◆ **-r**: 递归删除 (删除目录和目录内的所有文件)。
- ◆ **-f**: 强制删除, 不进行提示。
- ◆ **-v**: 输出说明文字。
- ◆ **file**: 要删除的文件。

使用举例:

```
rm -rf * # 强制删除当前目录下的所有文件 (包括目录)
```

3.2.2.7 find

find 命令可以在指定目录内搜索满足指定条件的文件, 搜索范围将遍历整个目录树, 基本用法如下:

```
find [path ...] [expression]
```

其各个参数的含义解释如下。

- ◆ **path**: 指定要搜索的目录, 如省略, 则为当前目录。
- ◆ **expression**: 表示搜索的条件, 符合条件的文件的路径将被显示出来, 如省略, 表示无条件, 所有文件的路径都被显示出来。

expression 有以下常用内容。

- ◆ **-name pattern**: 表示文件名与 **pattern** 匹配 (可使用通配符 *****, **?** 等)。
- ◆ **-iname pattern**: 类似于 **-name pattern**, 但不区分大小写。
- ◆ **-amin n**: 文件在 **n** 分钟前被访问过, **n** 是一个数字。
- ◆ **-atime n**: 文件在 **n** 天前被访问过, **n** 是一个数字。

- ◆ `-mmin n`: 文件内容在 `n` 分钟前被修改过, `n` 是一个数字。
- ◆ `-mtime n`: 文件内容在 `n` 天前被修改过, `n` 是一个数字。
- ◆ `-empty`: 文件是空的。
- ◆ `-type t`: 文件类型是 `t`, `t` 是一个代表文件类型的字符。

其中表示时间长度的 `n` 前面还可以加正号 `+` 表示大于 `n`, 加负号 `-` 表示小于 `n`。表示文件类型的字符可以是以下值: `b` 表示块设备文件, `c` 表示字符设备文件, `d` 表示目录, `p` 表示命名管道, `f` 表示普通文件, `l` 表示符号链接, `s` 表示套接字。

多个条件还可以用以下操作符进行逻辑运算。

- ◆ `!expr`: 表示对 `expr` 的值做逻辑非运算。
- ◆ `expr1 -a expr2`: 表示对 `expr1` 和 `expr2` 的值做逻辑与运算, 其中 `-a` 可省略。
- ◆ `expr1 -o expr2`: 表示对 `expr1` 和 `expr2` 的值做逻辑或运算。

使用举例:

```
find -mmin +30 -mmin -60 # 查找 60 分钟内 30 分钟前修改过的文件
find -name file* # 查找文件名前 4 个字母是 file 的文件
```

3.2.2.8 touch

`touch` 命令将文件的修改时间改为当前时间, 如果文件不存在, 将创建这个文件, 文件内容为空, 基本用法如下:

```
touch file ...
```

其参数的含义解释如下。

- ◆ `file`: 被修改 (或创建) 的文件。

3.2.3 文件内容相关

3.2.3.1 cat

`cat` 命令用于读取文件内容并输出, 基本用法如下:

```
cat [file ...]
```

其参数的含义解释如下。

- ◆ `file`: 被读取的文件, 如果省略, 则读标准输入。

3.2.3.2 more 和 less

`more` 命令和 `less` 命令的功能非常相似, 它们都能读取文件的内容并输出, 当文件显示的行数超过屏幕的行数时, 显示暂停并允许用户上下翻页。它们的基本用法如下:

```
more file ...
```

```
less file ...
```

其参数的含义解释如下。

◆ **file**: 被读取的文件。

它们都支持用 **Ctrl+F** 组合键向下翻页, 用 **Ctrl+B** 组合键向上翻页, 用 **q** 键退出。

3.2.3.3 head 和 tail

head 命令用于读取文件内容前面的部分并输出, 基本用法如下:

```
head [-c bytes] [-n lines] [file ...]
```

其各个参数的含义解释如下。

- ◆ **-c bytes**: 指定读取前 **bytes** 个字符, 如果在 **bytes** 前加一个负号, 则读取除最后 **bytes** 个字符以外的全部内容。
- ◆ **-n lines**: 指定读取前 **lines** 行内容, 如果在 **lines** 前加一个负号, 则读取除最后 **lines** 行之外的全部内容。
- ◆ **file**: 要读取的文件, 如省略, 则读标准输入。

tail 命令用于读取文件内容后面的部分并输出, 基本用法如下:

```
tail [-c bytes] [-n lines] [-f] [file ...]
```

其各个参数的含义解释如下。

- ◆ **-c bytes**: 指定读取后 **bytes** 个字符, 如果在 **bytes** 前加一个正号, 则读取除开始 **bytes** 个字符以外的全部内容。
- ◆ **-n lines**: 指定读取后 **lines** 行内容, 如果在 **lines** 前加一个正号, 则读取除开始 **lines** 行之外的全部内容。
- ◆ **-f**: 读到文件结束时不退出, 如果文件的内容动态增加, 则继续显示增加的部分。
- ◆ **file**: 要读取的文件, 如省略, 则读标准输入。

3.2.3.4 grep

grep 命令用于搜索并输出文件内容中包含指定模式的行, 基本用法如下:

```
grep [-i] [-r] pattern [file ...]
```

其各个参数的含义解释如下。

- ◆ **-i**: 忽略大小写。
- ◆ **-r**: 递归搜索, 即如果 **file** 是目录, 则搜索整个目录树下的所有文件。
- ◆ **pattern**: 一个基本的正则表达式, 用于匹配文件内容。
- ◆ **file**: 要搜索的文件, 如省略, 则搜索标准输入。

使用举例:

```
grep -r -i abcde . # 搜索当前目录下所有文件的内容
```

3.2.3.5 sort

sort 命令可以对文件的内容按行排序并输出,基本用法如下:

```
sort [-b] [-i] [-f] [-n] [-r] [file ...]
```

其各个参数的含义解释如下。

- ◆ **-b**: 忽略开始的空格。
- ◆ **-i**: 忽略不可打印的字符。
- ◆ **-f**: 忽略字母的大小写。
- ◆ **-n**: 将字符串转为数字后排序。
- ◆ **-r**: 反向排序,默认是从小到大。
- ◆ **file**: 被读取的文件,如省略,则默认为标准输入。

3.2.3.6 wc

wc 用于统计文件中的字符数、单词数、行数,基本用法如下:

```
wc [file ...]
```

其参数的含义解释如下。

- ◆ **file**: 被统计的文件,如省略,则统计标准输入。

3.2.4 压缩与解压缩

3.2.4.1 gzip 和 bzip2

gzip 命令和 **bzip2** 命令用于压缩和解压缩一个文件,它们使用两种不同的压缩算法,**bzip2** 的压缩率一般高于 **gzip**,但速度也要慢一些。它们的基本用法如下:

```
gzip [-d] [-v] [-1..9] file ...  
bzip2 [-d] [-v] [-1..9] file ...
```

其各个参数的含义解释如下。

- ◆ **-d**: 解压缩方式。
- ◆ **-v**: 输出说明文字。
- ◆ **-1..9**: 从 1 到 9 的数字,用于指定压缩率级别,级别越高,压缩率越大。
- ◆ **file**: 被压缩或解压缩的文件。

被 **gzip** 命令压缩过的文件自动加上 **.gz** 后缀,被 **bzip2** 命令压缩过的文件自动加上 **.bz2** 后缀,默认都会将原文件删除。解压缩的过程相反,文件的 **.bz**, **.bz**, **.bz2** 等后缀被去掉,作为

解压出来的文件名，原压缩文件则被删除。

3.2.4.2 tar

tar 命令可以将多个文件归档为一个文件，也可以从归档文件中提取出原文件。归档文件中记载了原文件的路径，因此可以保证提取出的文件相对位置不变。**tar** 命令还可以自动调用 **gzip** 命令或 **bzip2** 命令对归档后的文件进行压缩与解压缩，使用十分方便，因而成为了 Linux 操作系统上最常用的压缩与解压缩工具。它的基本用法如下：

```
tar [-C path] -c|-x|-t [-z|-j] [-v] -f target [file ...]
```

其各个参数的含义解释如下。

- ◆ **-C path**: 将从归档文件提取出的原文件放到 **path** 目录中，默认放到当前目录中。
- ◆ **-c**: 建立新的归档文件。
- ◆ **-x**: 从归档文件提取原文件。
- ◆ **-t**: 模拟提取文件操作，并不真正将文件提取出来。
- ◆ **-z**: 调用 **gzip** 进行压缩或解压缩。
- ◆ **-j**: 调用 **bzip2** 进行压缩或解压缩。
- ◆ **-v**: 输出说明文字。
- ◆ **-f target**: 指定归档文件的路径。
- ◆ **file**: 要归档的文件，提取文件时可以不指定。

使用举例：

```
tar -czvf foo.tar.gz foo # 将 foo 目录树归档为 foo.tar.gz，并用 gzip 压缩
tar -tzvf foo.tar.gz # 查看归档文件 foo.tar.gz 中包含的文件
tar -xzvf foo.tar.gz # 从归档文件 foo.tar.gz 中提取所有文件到当前目录
tar -C / -xjvf arm-linux-gcc.tar.bz2 # 提取文件到根目录
```

3.2.4.3 zip 和 unzip

zip 命令用于建立 **zip** 格式的压缩文件，基本用法如下：

```
zip [-r] file.zip file ...
```

其各个参数的含义解释如下。

- ◆ **-r**: 递归压缩，即如果 **file** 是目录，则对整个目录树进行压缩。
- ◆ **file.zip**: 压缩后的文件。
- ◆ **file**: 被压缩的文件。

unzip 命令与 **zip** 命令相反，可以将 **zip** 格式的压缩文件解压缩，基本用法如下：

```
unzip file.zip
```

其参数的含义解释如下。

- ◆ file.zip: 被解压缩的文件。

3.2.5 文件系统与磁盘

3.2.5.1 mount 和 umount

mount 命令可以将一个文件系统挂载到指定的目录中, 基本用法如下:

```
mount [-t vfstype] [-v] dev dir
```

其各个参数的含义解释如下。

- ◆ -t vfstype: 指定文件系统的类型是 vfstype, 省略则自动选择。
- ◆ -v: 输出说明文字。
- ◆ dev: 块设备文件。
- ◆ dir: 某个目录。

Linux 系统上的存储设备一般以块设备文件的形式呈现。如果设备上已经建立了文件系统(即格式化), 就可以被挂载到某个目录。挂载之后, 目录中原有的内容被替代为设备的根目录中的内容, 这样就可以访问设备中的文件了。

很多存储设备支持分区, 每一个分区又可以作为一个单独的存储设备来使用。这样的设备上一般都会有单独的一块空间用于存放分区信息表。**mount** 命令可以根据分区信息表的内容自动判断要挂载的文件系统的类型, 如果无法判断, 则必须指定文件系统的类型才能正确挂载。Linux 系统上常用的文件系统类型有 ext2, ext3 等, 但一般也能访问 Windows 系统的文件系统类型, 如 vfat 和 ntfs。

umount 命令用于卸载文件系统, 基本用法如下:

```
umount dir|dev
```

其各个参数的含义解释如下。

- ◆ dir: 已挂载了文件系统的目录。
- ◆ dev: 被挂载的块设备文件。

umount 命令与 **mount** 命令相反, 它可以从指定的目录卸载文件系统, 或将指定设备上的文件系统从目录中卸载。卸载之后, 此目录中的内容恢复到挂载前的状态。

挂载和卸载文件系统必须由具有 root 权限的用户操作。

使用举例:

```
mount -t vfat /dev/sda1 /mnt # 将 /dev/sda1 设备挂载到 /mnt 目录
umount /mnt # 卸载 /mnt 目录中所挂载的文件系统
```

3.2.5.2 df 和 du

df 命令可以查询系统中各个存储设备的使用状况, 基本用法如下:

```
df [-h]
```

其参数的含义解释如下。

- ◆ **-h**: 增强可读性, 使用 KB, MB, GB 等单位来显示存储空间的大小。

du 命令可以查询文件所占用的磁盘空间大小, 基本用法如下:

```
du [-c] [-s] [-h] [file ...]
```

其各个参数的含义解释如下。

- ◆ **-c**: 最后再输出所有文件占用空间的总和。
- ◆ **-s**: 对每个所列文件 (目录), 只输出一个占用空间的总和。
- ◆ **-h**: 增强可读性, 使用 KB, MB, GB 等单位来显示存储空间的大小。
- ◆ **file**: 要查询的文件, 默认是当前目录。

对于目录, **du** 查询到的是整个目录树占用的空间大小, 而不是目录本身的大小。

3.2.6 用户与权限

3.2.6.1 who

who 命令可以查询当前有哪些用户已经登录本系统, 一般不需要参数。

3.2.6.2 groups

groups 命令可以查询某个用户属于哪些用户组, 基本用法如下:

```
groups [user ...]
```

其参数的含义解释如下。

- ◆ **user**: 要查询的用户名, 如省略, 则查询当前用户。

3.2.6.3 passwd

passwd 命令用于修改用户密码, 基本用法如下:

```
passwd [-S] [-a] [-d] [user]
```

其各个参数的含义解释如下。

- ◆ **-S**: 查看指定用户的密码状态。
- ◆ **-a**: 与 **-S** 结合, 查看所有用户的密码状态。
- ◆ **-d**: 删除指定用户的密码。
- ◆ **user**: 指定要操作的用户, 如省略, 默认为当前用户。

普通用户只能操作自己的账号, **root** 用户可以对所有账号进行操作。

3.2.6.4 su

su 命令可以切换到其他用户，基本用法如下：

```
su [-l] [-p] [user]
```

其各个参数的含义解释如下。

- ◆ **-l**: 切换后为登录状态，如同用户直接登录的环境一样。参数中字母 **l** 可省略。
- ◆ **-p**: 保持环境变量不变。
- ◆ **user**: 指定切换到的用户，如省略，默认为 **root**。

切换用户后将开启一个新的 **Shell**，退出这个 **Shell** 就返回了原来的用户。

3.2.6.5 exit

exit 命令用于退出当前 **Shell**，一般不需要参数。

3.2.6.6 chown

chown 命令用于修改文件的所属用户和所属组，基本用法如下：

```
chown [-R] [-v] owner[:group] file ...
```

其各个参数的含义解释如下。

- ◆ **-R**: 递归修改，如果 **file** 是目录，则修改整个目录树内的所有文件。
- ◆ **-v**: 输出说明文字。
- ◆ **owner**: 新的所属用户。
- ◆ **group**: 新的所属组。
- ◆ **file**: 被修改的文件。

普通用户只能对自己所有的文件进行修改，**root** 用户可以对任何文件进行修改。
使用举例：

```
chown -R root:root mydoc
```

这条命令将 **mydoc** 目录树中的所有文件改为 **root** 用户及 **root** 组所有。

3.2.6.7 chmod

chmod 命令用于修改文件的权限标志，基本用法如下：

```
chmod [-R] [-v] mode file ...
```

其各个参数的含义解释如下。

- ◆ **-R**: 递归修改，如果 **file** 是目录，则修改整个目录树内的所有文件。
- ◆ **-v**: 输出说明文字。

- ◆ **mode**: 新的权限标志。
- ◆ **file**: 要修改的文件。

新的权限标志有以下两种表达方式。

- ◆ **相对方式**: 由三个字符组成。第一个字符从字母 **u**, **g**, **o**, **a** 中选取, 分别代表对所属用户、所属组、其他用户以及全部的权限标志位进行操作; 第二个字符是加号 **+**, 代表增加权限, 或者减号 **-**, 代表去除权限; 第三个字符为 **r**, **w**, **x** 之一, 分别表示读、写、执行权限。修改时将在原有权限的基础上增加或减少所指定的权限。
- ◆ **绝对方式**: 由三个数字组成。第一个数字代表所属用户权限, 第二个数字代表所属组权限, 第三个数字代表其他用户权限。数字的取值范围是 **0~7**, 将其转化为二进制数后, 从高位到低位的每一位分别代表读、写、执行权限。修改时文件的权限标志直接设成所指定的值, 不考虑原来的权限。

普通用户只能对自己所有的文件进行修改, **root** 用户可以对任何文件进行修改。

使用举例:

```
chmod 777 myfile # 将文件 myfile 的权限改为所有用户可读、可写、可执行
chmod 660 myfile # 将文件 myfile 的权限改为所属用户和组用户可读写, 不能执行
chmod g+x myfile # 给文件 myfile 增加组内用户可执行权限
```

3.2.7 进程管理

3.2.7.1 ps

ps 命令用于查看系统中的进程, 基本用法如下:

```
ps [-e] [-l]
```

其各个参数的含义解释如下。

- ◆ **-e**: 显示系统中的所有进程, 默认只显示本会话上的进程。
- ◆ **-l**: 使用长格式, 输出关于进程的更详细的信息。

3.2.7.2 kill

kill 命令用于向进程发送信号, 基本用法如下:

```
kill [-signal] pid ...
```

其各个参数的含义解释如下。

- ◆ **-signal**: 向进程发送 **signal** 信号, **signal** 可以是信号的编码或名称, 如省略, 默认发送 **TERM** 信号, 一般情况下将使进程退出。
- ◆ **pid**: 信号发送的目标进程的进程号。

通常用 **kill** 命令使某个进程退出。如果进程不响应 **TERM** 信号, 则可以发送 **KILL** 信号, 这

是一个进程不能忽略的信号。进程号可以通过 `ps` 命令查询得到。用以下命令可以查询能够发送的所有信号的列表：

```
kill -l
```

使用举例：

```
kill 948 # 向进程 948 发送 TERM 信号，使其退出
kill -9 948 # 向进程 948 发送 KILL 信号，使其退出，KILL 信号的编码为 9
```

3.2.7.3 init

`init` 命令可用于切换运行级别，基本用法如下：

```
init 0|1|6
```

其各个参数的含义解释如下。

- ◆ 0：切换到运行级别 0，代表关机。
- ◆ 1：切换到运行级别 1，代表单用户模式。
- ◆ 6：切换到运行级别 6，代表重新启动。

切换运行级别只能由 `root` 用户进行操作。

3.2.7.4 top

`top` 命令用于动态监视进程的运行状况，基本用法如下：

```
top [-d delay] [-p pid ...]
```

其各个参数的含义解释如下。

- ◆ `-d delay`：指定信息刷新的时间间隔。
- ◆ `-p pid`：指定要监视的进程，如省略，则为全部进程。

`top` 命令执行过程中按 `q` 键可退出。

3.2.8 系统信息

3.2.8.1 uname

`uname` 命令用于显示系统信息，基本用法如下：

```
uname [-a] [-r]
```

其各个参数的含义解释如下。

- ◆ `-a`：显示所有信息，默认只显示操作系统名称，即 `Linux`。
- ◆ `-r`：显示内核版本信息。

3.2.8.2 free

free 命令用于查询系统当前的内存使用状况，基本用法如下：

```
free [-b|-k|-m|-g] [-s delay]
```

其各个参数的含义解释如下。

- ◆ **-b**: 指定显示数据以字节为单位。
- ◆ **-k**: 指定显示数据以 KB 为单位。
- ◆ **-m**: 指定显示数据以 MB 为单位。
- ◆ **-g**: 指定显示数据以 GB 为单位。
- ◆ **-s delay**: 指定每隔 **delay** 秒数据更新一次，如省略，则只显示一次数据。

3.2.9 网络

3.2.9.1 ping

ping 命令用于测试网络连接是否正常，基本用法如下：

```
ping [-a] [-c count] [-i interval] [-s packetsize] destination
```

其各个参数的含义解释如下。

- ◆ **-a**: 收到响应时发出声音。
- ◆ **-c count**: 发送数据包的个数，默认为无限个。
- ◆ **-i interval**: 发送数据包的时间间隔，默认为 1 秒。
- ◆ **-s packetsize**: 数据包的大小，默认为 56 字节。实际发送的数据包还要加上 8 个字节的 ICMP 包头，成为 64 字节。
- ◆ **destination**: 目标的 IP 地址或主机名。

3.2.9.2 ifconfig

ifconfig 命令用于管理网络设备。首先可以用它对网络设备的信息进行查询，基本用法如下：

```
ifconfig [-a] [interface]
```

其各个参数的含义解释如下。

- ◆ **-a**: 显示所有设备的信息。如省略，则只显示处于启用状态的设备。
- ◆ **interface**: 网络设备的名称，此参数表示只显示指定的网络设备的信息。

查询操作可以由普通用户执行，而其他配置操作都需要 **root** 权限才能执行，包括启用和停用网络设备、设置 IP 地址、设置硬件地址等。

启用和停用网络设备的基本用法如下：

```
ifconfig interface up|down
```

其各个参数的含义解释如下。

- ◆ interface: 被启用或停用的网络设备名称。
- ◆ up: 启用。
- ◆ down: 停用。

设置 IP 地址的基本用法如下:

```
ifconfig interface address [netmask addr]
```

其各个参数的含义解释如下。

- ◆ interface: 要设置的网络设备的名称。
- ◆ address: 新的 IP 地址。
- ◆ netmask addr: 指定子网掩码为 **addr**, 如省略, 则根据 IP 地址的类型自动确定。

设置硬件地址的基本用法如下:

```
ifconfig interface hw class address
```

其各个参数的含义解释如下。

- ◆ interface: 要设置的网络设备的名称。
- ◆ hw class address: 指定网络设备的类型为 **class**, 硬件地址为 **address**。对于常用的以太网卡, 类型固定为 **ether**, 硬件地址就是 **MAC** 地址。

设置硬件地址时要求网络设备处于停用状态。

使用举例:

```
ifconfig eth0 down # 停用网络设备 eth0, eth0 一般就是第一块以太网卡
ifconfig eth0 hw ether 00:0c:29:60:5a:17 # 设置以太网卡 eth0 的 MAC 地址
ifconfig eth0 192.168.1.135 netmask 255.255.255.0 # 设置 IP 地址和掩码
```



ifconfig 命令所设置的地址并没有保存在配置文件中, 因此不能保持到主机重启以后。要想永久改变网络设置, 必须修改系统的网络配置文件, 可以用系统提供的专用配置软件来进行。

3.2.9.3 route

route 命令用于管理路由表。首先可以用它查询路由表, 基本用法如下:

```
route [-n]
```

其参数的含义解释如下。

- ◆ **-n**: 显示数字形式的 IP 地址, 如果省略, 将在有可能的情况下显示主机名。

查询操作可以由普通用户执行, 而删除和添加路由操作必须由 **root** 用户执行。

添加路由的基本用法如下：

```
route add [-net] target [netmask Nm] [gw Gw] [[dev] interface]
```

其各个参数的含义解释如下。

- ◆ `-net`：指定地址 `target` 代表一个网络，如省略或换成 `-host`，则地址 `target` 表示一台主机。
- ◆ `netmask Nm`：当 `target` 是网络时，指定其掩码为 `Nm`，是主机时无须指定。
- ◆ `gw Gw`：指定此路由的网关地址是 `Gw`。
- ◆ `dev interface`：指定此路由使用的网络设备是 `interface`，`dev` 可以省略。

删除路由的基本用法如下：

```
route del [-net] target [netmask Nm] [[dev] interface]
```

其各个参数的含义解释如下。

- ◆ `-net`：指定地址 `target` 代表一个网络，如省略或换成 `-host`，则地址 `target` 表示一台主机。
- ◆ `netmask Nm`：当 `target` 是网络时，指定其掩码为 `Nm`，是主机时无须指定。
- ◆ `dev interface`：指定此路由使用的网络设备是 `interface`，`dev` 可以省略。

使用举例：

```
route del default # 删除默认路由
route add default dev eth0 # 添加默认路由，使用设备 eth0
route add -net 192.168.1.0 netmask 255.255.255.0 gw 192.168.1.2 dev eth0
# 添加到网络 192.168.1.x 的路由，网关为 192.168.1.2，使用设备 eth0
```

3.2.9.4 dhclient

`dhclient` 命令实际上是 DHCP 协议的客户端，使用它可以动态分配 IP 地址，一般不需要参数。

3.2.9.5 ftp

`ftp` 是 FTP 协议客户端，它可以用来登录 FTP 服务器，并上传或下载文件。输入以下命令可以连接指定的服务器：

```
ftp [host] [port]
```

其各个参数的含义解释如下。

- ◆ `host`：主机名或者 IP 地址。
- ◆ `port`：端口号，可省略，默认为 21。

连接时可能需要用户名和密码。如果主机地址和端口号都省略，则直接进入 `ftp` 的交互界面，可以输入 `ftp` 命令进行操作。常用的一些 `ftp` 命令如表 3.1 所示。

表 3.1 常用 ftp 命令

命令	功能
help	查看命令帮助
open	连接 FTP 服务器
ls	查看远程目录的内容列表
cd	切换远程工作目录
get	下载文件
put	上传文件
ascii	切换为 ASCII 传输模式
binary	切换为二进制传输模式
close	断开连接
bye 或 quit	退出 ftp 界面

3.2.9.6 wget

wget 是一个强大的下载工具，支持 HTTP 和 FTP 等协议。它的基本用法如下：

```
wget [-r] [-k] [-c] [-O file] [URL ...]
```

其各个参数的含义解释如下。

- ◆ -r：递归下载，解析所下载的 HTML 文件中的超链接目标并下载。
- ◆ -k：递归下载时，将所下载的 HTML 文件中的超链接修改为指向本地文件。
- ◆ -c：断点续传，继续下载上次未下载完成的文件。
- ◆ -O file：下载的文件名改为 file，如省略，将自动通过网址确定。
- ◆ URL：要下载的网址。

3.3 Shell 使用进阶

本节将介绍 Shell 的一些高级特性，通过学习这些特性可以进一步加深对 Linux 操作系统的理解，合理地应用这些功能将极大地提高 Linux 下的开发效率。

3.3.1 重定向

Shell 在启动一个程序之前，能够以指定的文件描述符打开一些文件，被打开的文件在程序内可以直接进行读写，无须再次打开。通常 Shell 会默认打开三个设备文件，如表 3.2 所示。

标准输入一般是终端键盘，用于读取用户输入的信息；标准输出一般是控制台屏幕，用于输出信息；标准错误输出一般也是控制台屏幕，用于输出调试与错误信息。

Shell 允许在输入命令时指定要打开的文件和所使用的文件描述符，指定的内容可以覆盖默认的内容，称为重定向。

表 3.2 Shell 默认打开的文件

名称	文件描述符	对应的 C 程序 FILE * 指针
标准输入	0	stdin
标准输出	1	stdout
标准错误输出	2	stderr

指定打开文件的方式如表 3.3 所示。其中 `n` 表示要使用的文件描述符，如果省略，则使用默认值。

表 3.3 重定向的书写格式

写法	作用	描述符默认值
<code>n> file</code>	以只写方式打开文件 <code>file</code> ，文件原有内容将被清空	1
<code>n< file</code>	以只读方式打开文件 <code>file</code>	0
<code>n>> file</code>	以只写方式打开文件 <code>file</code> ，新写入内容附加在原文件内容之后	1
<code>n<> file</code>	以可读、可写方式打开文件 <code>file</code>	0

`file` 除了可以写文件名之外，还可以写符号 `&` 加一个已打开的文件描述符，表示将新的文件描述符重定向到已打开的文件。

输出重定向经常用来把本应显示在屏幕上的信息写入文件，例如：

```
echo abcde > myfile
```

`echo` 命令既是内置命令又是外部命令，它的作用是把命令参数原样显示在标准输出上，因此以上命令的作用就是将 `abcde` 这行内容写入文件 `myfile`。

有时希望将标准输出和标准错误输出的信息全部写入同一个文件中，例如：

```
make > myfile 2> myfile
```

这种做法不能达到目的，因为两个输出的内容在同一个文件中相互覆盖，正确的做法是将标准输出重定向到文件，再将标准错误输出重定向到标准输出，如：

```
make > myfile 2>&1
```

如果写为：

```
make 2>&1 > myfile
```

则 `myfile` 中只保存了标准输出的内容，标准错误输出仍然显示在屏幕上。这是因为 Shell 按从左到右的顺序处理重定向，上述命令先将标准错误输出重定向到原来的标准输出，即控制台屏幕，然后再将标准输出重定向到文件 `myfile`。

事实上，Shell 支持一种更简便的写法，也能达到同样目的：

```
make &> myfile
```

输入重定向常用来让一些有交互界面的程序自动执行既定的命令，如：

```
ftp -n < command.txt # 启动 ftp 客户端并从 command.txt 文件中读入命令执行
# 这里的 -n 参数用于阻止自动登录
```

3.3.2 管道

将两条命令用竖线 | 连接起来，这样它们会在同一个进程组内执行，前一条命令的标准输出将通过管道的方式传递给后一条命令的标准输入。

管道最基本的用法是与文件内容相关命令结合对输出到屏幕的信息进行过滤，如：

```
ls -l | more # 列出当前目录下的文件，如果文件数过多，则允许上下翻页
ls | wc # 统计当前目录下的文件数
ps | grep bash # 查询进程信息，只显示包含 bash 字符串的行
```

输出重定向后屏幕上不再显示信息。如果希望两个输出的内容既显示在屏幕上，又记录在文件中，则可以利用命令 **tee**，它能够把从标准输入读入的内容同时输出到标准输出和多个文件中，基本用法如下：

```
tee [-a] file ...
```

其各参数的含义解释如下。

- ◆ **-a**：添加新内容到文件的末尾，如果省略，则覆盖文件原来的内容。
- ◆ **file**：要写入的文件。

因此可以使用管道结合 **tee** 命令来达到前述目的，如：

```
make 2>&1 | tee myfile
```

以上命令将 **make** 命令的标准错误输出重定向到标准输出，合并后的结果再通过管道传送给 **tee** 命令，实现了同时在屏幕上显示并且记录到文件的功能。

管道还经常与另外一条命令 **xargs** 结合使用。**xargs** 命令的功能比较特殊，它的参数可以是一条 **Shell** 命令，产生的作用是从标准输入读入数据作为这条命令的参数，并执行这条命令。用它与管道组合，就可以依据其他命令产生的结果去做指定的操作，例如：

```
find -type f | xargs rm -f # 删除当前目录树下的所有普通文件
```

事实上，**find** 命令本身就可以对找到的文件进行删除操作，因此上述命令可以写为：

```
find -type f -delete # -delete 表示对找到的文件进行删除操作
```

执行这条命令时省去了在两个进程间传递数据，因而更高效。

3.3.3 变量与替换

3.3.3.1 变量的定义与使用

Shell 中可以定义变量，如：

```
ABC=hello # 定义变量 ABC, 值为 hello
```



定义变量时等号两边不可有空格, 否则会被 Shell 错误解析。

定义的变量可以在输入其他命令时使用, 如:

```
echo ${ABC} # ${ABC} 将被变量 ABC 的值替换, 相当于 echo hello
```

其中的大括号在不引起混淆的情况下可以省略。

事实上, 这种替换可以出现在命令的任何位置上, 甚至是参数的一部分, 如:

```
LS=ls # 定义变量 LS, 值为 ls
${LS} -l # 相当于 ls -l
${LS} ${LS}* # 相当于 ls ls*, 列出当前目录下以 ls 开头的文件
```

已定义的变量可用 `unset` 命令取消定义, 如:

```
unset ABC LS # 取消变量 ABC 和 LS 的定义
```

3.3.3.2 命令替换

除了变量替换之外, Shell 还支持另外一种替换, 称为命令替换, 如:

```
echo $(uname) # 显示 Linux, $(uname) 被命令 uname 的执行结果替换
```

注意这时圆括号内的命令是在一个子 Shell 中执行的, 执行完毕就退出, 因此不会改变当前 Shell 的环境设置。

3.3.3.3 三种引号

当命令的某个参数中包含空格 (例如文件名中有空格) 时, 就需要用单引号包围起来, 如:

```
rm -rf 'My Document' # 删除 My Document 目录树
```

这时也可以使用双引号, 区别在于, 双引号内出现的变量会被变量值替换, 单引号则保持原样不被替换, 如:

```
ABC=hello
echo "string is ${ABC}" # 显示 string is hello
echo 'string is ${ABC}' # 显示 string is ${ABC}
```

Shell 中还可以使用反引号, 实际上就是命令替换, 如:

```
echo `uname` # 等价于 echo $(uname)
```

3.3.3.4 转义字符

文件名中包含空格时, 还有另外一种处理方案, 即使用转义字符, 如

```
rm -rf My\ Document # 使用反斜杠转义后, 空格被解析为字符串的一部分
```

因为反斜杠、单引号、双引号、反引号以及符号 `$` 等都有特殊含义, 如果让这些字符直接出现在命令行中, 则有可能需要转义。

3.3.3.5 通配符

在命令中输入文件名时可以使用通配符。通配符有两种: 符号 `?` 用于匹配一个字符, 符号 `*` 用于匹配任意长度的字符串。**Shell** 在处理含有通配符的文件名时, 将其展开成所有匹配的文件名, 文件名之间用空格隔开, 如:

```
echo * # 显示当前目录下所有文件的名称  
rm test? # 删除当前目录下所有名称为 test 加一个字符的文件
```

3.3.4 环境变量

Shell 在启动可执行程序时, 会将一些变量的名称和值作为参数传递给此程序, 这些变量称为环境变量, 很多命令都会根据环境变量的值改变自己的行为。变量经导出之后就成为环境变量, 如:

```
ABC=hello  
export ABC # 导出变量 ABC
```

也可以同时定义并导出变量, 如:

```
export ABC=hello DEF=world # 定义并导出环境变量 ABC 和 DEF
```

不带参数的 `export` 命令可用于查询所有环境变量。

在一条 **Shell** 命令之前可以加上变量的定义, 这种变量只对此命令有效, 并且不影响 **Shell** 中已有的变量, 命令执行后即消失, 这里称之为临时环境变量, 如:

```
LANG=zh_CN.UTF-8 ls --help # 将语言设置为中文以后查看 ls 命令的帮助
```

Shell 中有一些预定义的重要的环境变量, 列举如下。

- ◆ **HOME**: 当前用户的家目录。
- ◆ **USER**: 当前用户。
- ◆ **PWD**: 当前目录。
- ◆ **PATH**: **Shell** 搜索可执行文件的路径, 多条路径用冒号隔开。

当用户输入命令要执行时, 如果不是内置命令, **Shell** 将在 **PATH** 环境变量所指示的目录中查找。通常 **PATH** 变量设置如下:

```
PATH=/usr/local/bin:/usr/bin:/bin
```

这些目录中包含了大多数常用命令, 故使用这些命令时直接输入可执行文件的名称就可以, 无须指明路径。如果输入命令时用的是可执行文件的路径, 且路径中至少包含一个斜杠 (否则无法判断是路径还是只有文件名), 则 **Shell** 不在 **PATH** 变量所指示的目录中查找, 而是直接启动执行相应的程序。一般来说, **PATH** 变量的设置不应该包含当前目录在内, 因此如果要执行当前目录下的

程序，可使用相对路径，如：

```
./my_prog # my_prog 是当前目录下的一个可执行文件
```

3.3.5 脚本

3.3.5.1 脚本的执行

Shell 命令可以事先写在一个文件中，使用时 Shell 读取文件中的命令逐条解释执行，这种文件被称为脚本。执行脚本的方法与执行其他程序一样，如：

```
./setenv.sh # setenv.sh 是当前目录下的一个脚本
```



被执行的脚本必须先赋予可执行权限，可用 `chmod +x` 命令实现。

以这种方式执行脚本时，其中的命令是在一个子 Shell 中执行的。子 Shell 继承了父 Shell 的环境变量，但无法修改它们，或者说所做的修改仅对子 Shell 有效。如果要使用脚本修改环境变量，则必须让脚本在当前 Shell 中执行，这一点可用以下命令实现：

```
source setenv.sh # 读入 setenv.sh 文件中的命令，在当前 Shell 中执行
```

`source` 是内置命令，用途是读取文件的内容，并在当前 Shell 中逐条执行。这种方式执行的脚本无须可执行权限。`source` 命令可缩写为一个小数点，如：

```
. setenv.sh
```

3.3.5.2 启动脚本

系统启动最后会执行脚本 `/etc/rc.local`，它是在服务 `/etc/init.d/rc.local` 中被调用的，可以在这里做一些系统级的初始化工作。

`bash` 启动时也会自动执行一些脚本，如表 3.4 所示。

表 3.4 bash 启动脚本

脚本	说明
<code>/etc/profile</code>	全局登录脚本，任何用户登录时都执行
<code>/etc/bash.bashrc</code>	全局 <code>bash</code> 脚本，任何用户启动非登录的交互式 <code>bash</code> 时执行
<code>~/profile</code>	用户登录脚本，放在用户家目录下，此用户登录时执行
<code>~/bashrc</code>	用户 <code>bash</code> 脚本，放在用户家目录下，此用户启动非登录的交互式 <code>bash</code> 时执行

因此当一个用户登录时，`bash` 会顺序执行两个脚本：`/etc/profile` 和 `~/profile`。如果用户只是开启一个新的 `bash` 而不登录（例如使用 `su` 命令切换用户或在已登录的图形界面中打开新的终端窗口），则顺序执行 `/etc/bash.bashrc` 和 `~/bashrc`。

要注意这些都是交互式 `bash` 才有的行为。如果启动的 `bash` 不是交互式的（例如执行脚本时启动的子 Shell），则不执行这些脚本。



在很多 Linux 发行版上，`~/.profile` 中都会调用 `~/.bashrc`。

3.3.6 脚本编程

一般来说，脚本的第一行书写如下：

```
#!/bin/sh
```

Shell 将符号 `#` 后面的部分认为是注释，但这一行是有实际作用的，感叹号 `!` 后面的部分指明了要使用 `sh` 作为执行脚本的解释器。如果脚本中包含了 `bash` 特有的语法，则应该使用 `#!/bin/bash`。

脚本中的每一行是一条命令，多条命令也可以写在同一行，用分号 `;` 隔开。同其他编程语言一样，脚本也支持分支结构和循环结构，并且可以定义函数。

一、分支结构

Shell 支持分支结构，其中 `if` 分支结构的语法如下：

```
if list; then list; [ elif list; then list; ] ... [ else list; ] fi
```

这里各个 `list` 都代表一条或多条命令，方括号包围的部分为可选内容。执行流程是：先执行 `if` 子句中的命令，如果执行成功，则执行 `then` 子句中的命令，执行完毕后流程结束，转移到 `fi` 后面去执行；否则将从后面的第一个 `elif` 或 `else` 子句开始执行，依此类推。

事实上，每个 Linux 程序运行结束后都会给 Shell 一个返回值，这个值为 `0` 表示执行成功，其他值均表示失败。

`if` 及 `elif` 子句中的命令通常是用于判断的命令 `test`。`test` 命令的参数是一个表达式，表达式可以包含以下基本内容。

- ◆ `-n STRING`：字符串 `STRING` 的长度不是 `0`，其中 `-n` 可以省略。
- ◆ `-z STRING`：字符串 `STRING` 的长度是 `0`。
- ◆ `STRING1 = STRING2`：字符串 `STRING1` 与字符串 `STRING2` 相同。
- ◆ `STRING1 != STRING2`：字符串 `STRING1` 与字符串 `STRING2` 不同。
- ◆ `-b FILE`：文件 `FILE` 存在且是一个块设备文件。
- ◆ `-c FILE`：文件 `FILE` 存在且是一个字符设备文件。
- ◆ `-d FILE`：文件 `FILE` 存在且是一个目录。
- ◆ `-e FILE`：文件 `FILE` 存在。
- ◆ `-f FILE`：文件 `FILE` 存在且是一个普通文件。
- ◆ `-p FILE`：文件 `FILE` 存在且是一个命名管道。
- ◆ `-s FILE`：文件 `FILE` 存在且大小不为 `0`。
- ◆ `-r FILE`：文件 `FILE` 存在且可以读取。
- ◆ `-w FILE`：文件 `FILE` 存在且可以写入。
- ◆ `-x FILE`：文件 `FILE` 存在且可以执行。

表达式可以进行逻辑操作，形成一个新的表达式，如：

```
test ! -e file # 如果 file 不存在则返回成功，! 表示逻辑非
test -f file -a -x file # 如果 file 存在且可执行则返回成功，-a 表示逻辑与
test -b file -o -c file # 如果 file 存在且是设备则返回成功，-o 表示逻辑或
```

`test` 命令又经常写为左方括号 `[`，`[` 也是一个命令，功能与 `test` 相同，但是要求最后一个参数是右方括号 `]`，以求形式上的完整，如：

```
if [ -e myfile ]; then # 如果文件 myfile 存在
    echo File myfile exist! # 显示 File myfile exist!
fi
```

二、循环结构

Shell 也支持循环结构，其中 `for` 循环的一种语法如下：

```
for var in str1 str2 str3 ... ; do list ; done
```

这里 `var` 是循环变量。执行过程是：`var` 的值依次设为 `str1`，`str2`，`str3` 等，每设置一次值，执行一次 `list` 中的命令。举例如下：

```
for file in *.txt; do # 变量 file 的值依次设为当前目录下各个文本文件的名称
    cat ${file} # 输出文件 $file 的内容
done
```

三、函数

脚本中支持定义函数，例如：

```
print3 () {
    echo $1 $2 $3 # 显示第一、第二、第三个参数的值
}
```

其中符号 `$` 加一个数字将替换为调用函数时相应位置上的参数，`$*` 则会替换为所有参数。定义函数后就可以调用它：

```
print3 embedded linux program # 将显示 embedded linux program
```

执行脚本时也可以像执行其他程序一样传递命令行参数，在脚本中引用参数与在函数中引用参数的形式一样。需要注意的是，脚本中的函数引用的参数是调用函数时传递的参数而不是执行脚本时传递的参数。

3.3.7 作业管理

Linux 内核的进程管理机制使得 `bash` 可以实现作业管理。正在执行的每一条命令都是一个作业，每个作业有一个编号，可用 `jobs` 命令查看。`jobs` 命令的用法如下：

```
jobs [-l|-p] [job_id ...]
```

其各个参数的含义解释如下。

- ◆ `-l`: 列出作业的更详细信息。
- ◆ `-p`: 只列出作业对应的进程组组长的进程号。
- ◆ `job_id`: 指定的作业编号, 可省略, 默认是当前 Shell 中的所有作业。

用管道连接的多个程序启动后是一个作业, 尽管可能有多个进程在执行, 作业与进程并不是一一对应的关系。

一般情况下, 执行一个作业时, Shell 将等待其运行结束, 不再显示命令提示符, 因而无法再输入新的命令, 所输入的信息可被作业中的进程读取, 这种情况称为前台执行。Shell 还支持后台执行, 即作业在执行过程中, Shell 继续显示新的提示符, 接受用户的输入, 而作业中的进程则无法得到输入。后台执行的方法是在命令后加上一个符号 `&`, 如:

```
./myprog & # 后台方式执行当前目录下的 myprog 程序
```

当作业在前台执行时, 可按 `Ctrl+Z` 组合键向其主进程发送 `STOP` 信号, 进程对信号的默认处理是进入停止状态, Shell 恢复对终端的控制。最后一个停止的作业被 Shell 标记为当前作业, 如果当前作业执行完毕退出, 则上一个停止的作业就成为当前作业。

一个作业由符号 `%` 加它的编号或者启动命令来代表, 可称为作业的名称。作业的编号是唯一的, 而启动命令却有可能重复, 这时如果用启动命令作为作业的名称, Shell 会给出错误信息。如果只写一个 `%`, 则代表当前作业。

可以在 Shell 中输入命令改变作业的运行状态, 如:

```
bg %./myprog # 使作业在后台继续运行
fg %./myprog # 使作业在前台继续运行
```

使用 `bg` 和 `fg` 命令时, 作业名称上的符号 `%` 可以省略。如果省略作业名称, 则改变的是当前作业的状态。



当有作业在前台运行时, 尽管用户可以输入, 但不能被 Shell 读取。

还有一种更简单的用法如下:

```
%./myprog # 直接输入作业名, 使之在前台继续运行
%./myprog & # 输入作业名并加上符号 &, 使之在后台继续运行
```

3.4 Debian 5.0 的安装与使用

Debian 为基于 GNU/Linux 的众多发行版之一。它自从问世以来, 就以其严谨的软件包管理体系及良好的稳定性而被推崇。2009 年 2 月, Debian 发布了 5.0 稳定版本, 代号为 `lenny`。这一版本使用了 Linux 2.6.26 内核及各种较新版本的软件, 是 Linux 下进行开发工作的理想平台。

3.4.1 安装 Debian 5.0

Debian 5.0 的基本安装只需要使用其第一张 CD 光盘。从光盘启动后可以选择图形界面或命令行界面进行安装。安装过程中需要输入新的 root 用户密码,并建立一个新用户,其他配置均可以使用其默认值。安装完毕后重新启动,进入 Debian 桌面环境,如图 3.3 所示。



图 3.3 Debian 5.0 桌面



安装时会进行软件源扫描,根据网络状况,这一步可能会花很长时间。如果断开网络连接再进行安装,就可以跳过这一步,软件源可以等系统安装好以后手动设置。

Debian 配置了一些基本的图形界面管理工具,但多数情况下还是需要使用命令行界面进行操作。下面将只介绍命令行界面下的操作。

3.4.2 Debian 5.0 的基本操作

3.4.2.1 使用 sudo

在继续本节之前,首先介绍一个命令 `sudo`。该命令可以暂时以其他用户身份(默认为 root 用户)执行外部命令,执行完毕后返回原来的用户,如:

```
sudo cat file-of-root
```

这条命令以 root 用户身份读文件 `file-of-root` 的内容并输出。注意 `sudo` 不能执行内部命令,如 `cd`, `export`, `source` 等。

配置文件 `/etc/sudoers` 可以决定哪些用户能够使用 `sudo` 命令,编辑该文件,在其中加入下一行:

```
%sudo ALL=NOPASSWD: ALL
```

这行配置表示 `sudo` 组的用户执行 `sudo` 命令时不需要密码。然后将要使用 `sudo` 命令的用

户（如 **cjhy**）加入 **sudo** 组：

```
adduser cjhy sudo
```

注意这条命令的执行本身就需要 **root** 权限。

默认情况下，使用 **sudo** 切换到其他用户身份时并不是登录状态，只设置了基本的环境变量，故其使用有一定的局限性。

下文中提到的命令，凡是没有 **sudo** 也没有特殊说明的，表示这条命令的执行不需要 **root** 权限。

3.4.2.2 使用 **apt-get**

apt-get 是 **Debian** 的软件包管理工具，可用来从 **Debian** 的软件源下载、安装、升级软件。以下所列是它的基本用法，其中 **foo** 表示软件的名称。

安装软件包 **foo**：

```
sudo apt-get install foo
```

如果由于依赖关系破损而安装失败，可尝试用以下命令修复：

```
sudo apt-get -f install
```

如果只下载软件包 **foo** 而不安装，则加上 **-d** 参数：

```
sudo apt-get -d install foo
```

下载后的软件包放在 **/var/cache/apt/archives/** 目录下。

卸载软件包 **foo**，保留配置文件：

```
sudo apt-get remove foo
```

卸载软件包 **foo**，同时清除配置文件：

```
sudo apt-get purge foo
```

自动卸载已经不需要的软件包：

```
sudo apt-get autoremove
```

从磁盘缓存中清空已下载的软件包：

```
sudo apt-get clean
```

更新软件目录：

```
sudo apt-get update
```

升级已安装软件到最新：

```
sudo apt-get upgrade
```

如果要增加软件源，可编辑文件 **/etc/apt/sources.list**，按以下格式增加一行：

```
deb http://ftp.debian.org/debian lenny main non-free
```

这个源包含了用于 Debian 5.0 (lenny) 的主要软件包。

3.4.2.3 deb 包管理

Debian 使用 deb 格式的软件包，管理软件包的工具为 **dpkg**，以下所列是它的基本用法，其中 **foo** 表示软件的名称，**foo-1.0.0.deb** 表示对应的软件包的名称。一般来说，软件包的名称由软件名称、减号、版本号加后缀 **.deb** 组成。

安装软件包 **foo-1.0.0.deb**：

```
sudo dpkg -i foo-1.0.0.deb
```

卸载软件 **foo**：

```
sudo dpkg -r foo
```

查询软件 **foo** 的基本信息：

```
dpkg -s foo
```

上面命令是针对已安装的软件的，如果是未安装的，则用如下命令：

```
dpkg --info foo-1.0.0.deb
```

查看软件 **foo** 包含的文件列表：

```
dpkg -L foo
```

上面命令是针对已安装的包的，如果是未安装的，则用如下命令：

```
dpkg --contents foo-1.0.0.deb
```

如果要改变软件的安装目录，使用选项 **-instadir**，如：

```
sudo dpkg -i --instadir=/usr/local foo-1.0.0.deb
```

实际上，**apt-get** 也是使用 **dpkg** 来管理软件包的。

3.4.2.4 服务配置

目录 **/etc/init.d** 下面有每个服务对应的脚本，用来控制服务的启动和停止。下面以服务 **dbus** 为例说明这些脚本的用法。

启动 **dbus** 服务：

```
sudo /etc/init.d/dbus start
```

停止 **dbus** 服务：

```
sudo /etc/init.d/dbus stop
```

重启 **dbus** 服务：



```
sudo /etc/init.d/dbus restart
```

查看服务 `dbus` 的状态：

```
/etc/init.d/dbus status
```

所有开机自启动的服务放在目录 `/etc/rc?.d`（`?` 是当前运行级别，默认为 2，可用 `runlevel` 命令查看）下面，是一个指向上述脚本的符号链接，其名字必须是 `S` 加两位数字加服务名，如服务 `dbus` 的名字为 `S12dbus`。如果不希望它自动启动，只需将其名字改为 `K` 加两位数字开头，新的两位数字是原两位数字与 100 的差，如：

```
sudo mv /etc/rc2.d/S12dbus /etc/rc2.d/K88dbus
```

事实上，系统采用如下的方法管理服务的启动和停止：当进入某运行级别时，先以 `stop` 为参数执行相应目录下 `K` 打头的脚本，然后以 `start` 为参数执行 `S` 打头的脚本，执行的顺序与两位数字有关，数字越小越先执行。

3.4.3 常用软件的安装与使用

3.4.3.1 服务器软件

在嵌入式开发中常用的服务器软件包括 `nfs`，`tftp`，`ssh`，`samba` 等。

一、nfs

`nfs` 服务用于与 Linux 机器间共享文件夹。在嵌入式开发中通常用它来支持目标机的根文件系统。用以下命令可在主机上安装 `nfs` 服务：

```
sudo apt-get install nfs-kernel-server
```

在它的配置文件 `/etc/exports` 中加入以下一行将以可写权限共享指定的目录，这是进行嵌入式开发时常用的设置：

```
/usr/local/arm/sysroot *(rw, sync, no_root_squash, no_subtree_check)
```

这行配置将 `/usr/local/arm/sysroot` 作为共享目录。修改配置后，用以下命令使其生效（也可以重启 `nfs` 服务）：

```
sudo exportfs -arv
```

用类似于下面的命令可以挂载网络上共享的 `NFS` 目录：

```
sudo mount.nfs 192.168.198.129:/usr/local/arm/sysroot /mnt
```

这条命令将 192.168.198.129 主机上的 `/usr/local/arm/sysroot` 目录挂载到 `/mnt` 目录。通常用这个方法检查目录共享是否正常。

二、tftp

`tftp` 是一种基于 `UDP` 的简单文件传输协议。在嵌入式开发中，通常在 `bootloader` 阶段用这种协议下载内核映像等文件到目标机内存。用以下命令可在主机上安装 `tftp` 服务：



```
sudo apt-get install tftpd
```

tftpd 是 inetd 的一部分，其配置文件是 /etc/inetd.conf。安装 tftpd 之后，这个文件中应该有以下一行：

```
tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd /srv/tftp
```

这行配置的最后一项说明它默认使用的目录是 /srv/tftp，将要下载的文件置于这个目录内即可生效。

三、ssh

ssh 服务用于支持加密方式的远程登录，并支持通过 sftp 传输文件以及使用 scp 命令在不同主机间复制文件。用以下命令可在主机上安装 ssh 服务：

```
sudo apt-get install openssh-server
```

安装 ssh 服务的同时也安装了相关的一些其他工具。用 ssh 客户端可以远程登录另一台已启动 ssh 服务的主机，如：

```
ssh root@192.168.198.129
```

其中 root 是登录时使用的用户名，@ 符号后面的部分为远程主机地址，登录后就可以用 Shell 命令直接操作这台主机。

用 scp 命令可以在两台主机间复制文件，如：

```
scp cjhy@192.168.198.129:file-to-copy .
```

这条命令把目标主机上的文件 file-to-copy 复制到本地的当前目录下。目标主机上的文件名如果用了相对路径则是相对于所指定用户的家目录。当然也可以从本地复制文件到远程主机上。

用 sftp 可以如同 ftp 那样登录到远程主机，并上传或下载文件，如：

```
sftp cjhy@192.168.198.129
```

所用的命令与 ftp 工具类似。

四、samba

samba 服务用于与装 Windows 系统的主机间共享目录。用以下命令可安装 samba 服务：

```
sudo apt-get install samba
```

主机上的用户并不会自动成为 samba 账号，可用 smbpasswd 进行添加：

```
sudo smbpasswd -a cjhy
```

这条命令将用户 cjhy 添加为 samba 账号，以后在另外一台 Windows 机器上就可以用这个账号访问共享的目录。添加时需要给出 samba 账户密码，这个密码就是 Windows 机器访问本机时用的密码。

删除账号则可用以下命令：

```
sudo smbpasswd -x cjhy
```



`smbpasswd` 也可以用来修改 `samba` 密码，如：

```
smbpasswd
```

这条命令只能修改当前用户的密码，如欲修改其他用户密码则需要 `root` 权限：

```
sudo smbpasswd cjhy
```

共享目录的设置在文件 `/etc/samba/smb.conf` 中。`Debian 5.0` 默认已将用户的家目录共享，由这个文件中的以下几行控制：

```
[homes]
comment = Home Directories
browseable = no
read only = no
create mask = 0644
directory mask = 0755
```

按以上配置，用户的家目录将以可读可写的方式共享。

3.4.3.2 工具软件

嵌入式开发中常用的工具软件有 `kermit`，`vi` 等。这里介绍的软件不一定是必须安装的，或者可以安装它们的替代品。

一、kermit

`kermit` 是一个终端软件，嵌入式开发中用于从串口输入输出数据，作为目标机的控制台使用，可用以下命令安装：

```
sudo apt-get install ckermit
```

安装后使用以下命令启动 `kermit`：

```
kermit
```

启动后将显示如下提示符：

```
(/home/cjhy/) C-Kermit>
```

此时就可以输入 `kermit` 的命令了。以 `HY2410A` 开发板为例，将开发板用串口线连接到主机的第一个串口，然后在 `kermit` 界面中输入如下命令：

```
set line /dev/ttyS0 # 设置串口设备为 /dev/ttyS0
# 如果是 USB 转换的串口，则可能是 /dev/ttyUSB0
set serial 8N1 # 设置串口传输比特位为 8，校验位无，停止位为 1
set speed 115200 # 设置串口波特率为 115200b/s
set flow-control none # 设置流控制为无
set carrier-watch off # 设置载波监测为关闭
connect # 连接
```

连接之后 `kermit` 就成为开发板的控制台。`connect` 命令可缩写为 `c`，在连接状态中先按



Ctrl+\ 组合键，再按 c 键可断开连接回到 `kermit` 界面。用 `exit` 命令或 `quit` 命令可退出 `kermit`。

`kermit` 启动时会自动读取家目录下的 `.kermrc` 文件中的命令执行，可将上述命令放入此文件中，这样启动 `kermit` 时就能自动设置并连接开发板。也可以直接指定 `kermit` 启动时的脚本文件，如：

```
kermit hy2410.kermit # 启动 kermit 并从文件 hy2410.kermit 中读取命令执行
```

二、vi

`vi` 是 Linux 上常用的编辑器。在很多发行版上，`vi` 都是指向它的改进版 `vim` 的符号链接。Debian 5.0 自带的 `vim` 编辑器是简化版本，可用以下命令安装 `vim` 的完整版：

```
sudo apt-get install vim
```

`vim` 启动时要执行脚本 `/etc/vim/vimrc`，在这个文件中增加以下一行可打开语法高亮功能以利于阅读和编写程序：

```
syntax on
```

三、patch 与 diff

`patch` 工具用于给源码打补丁，而 `diff` 工具用来比较两个文件的差异，生成的结果可作为补丁文件使用。这两个工具一般已包含在默认安装选项中，如果未安装，可用以下命令安装：

```
sudo apt-get install patch diff
```

对单个文件打补丁的用法如下：

```
patch file-to-patch < patch-file # 对 file-to-patch 打补丁 patch-file
```

默认情况下 `patch` 工具从标准输入读补丁内容，所以要使用输入重定向。

对一个目录树内的文件打补丁的一般用法如下：

```
patch -d linux-2.6.30 -p1 < hy2410.patch
```

`-d` 参数用于指定打补丁的路径起点。因为补丁文件中都记录了要操作的文件的路径，所以只要指定一个正确的起点即可。如果不指定，默认为当前目录。`-p1` 参数指定在使用补丁文件中的路径前要将第一层目录去掉，这是因为制作补丁时的路径往往包含被打补丁的目录本身。

使用 `patch` 工具时加上 `-R` 参数可以反向打补丁，即将打过补丁的文件内容恢复到打补丁之前。

实际上，补丁就是一个文本文件，其中记录了两个文件的差异，这个文件可用 `diff` 工具来生成，如：

```
diff old-file new-file > new-file.diff
```

以上命令将文件 `old-file`，`new-file` 的差异记录在文件 `new-file.diff` 中，如果对 `old-file` 打上 `new-file.diff` 这个补丁，则 `old-file` 的内容就会跟 `new-file` 一致。`diff` 工具默认输出到标准输出上，因此要使用输出重定向。

比较两个目录并生成补丁的命令如下：

```
diff -Nur linux-2.6.30.orig linux-2.6.30 > hy2410.patch
```

以上命令将目录 `linux-2.6.30.orig` 和 `linux-2.6.30` 中的所有文件的差异记录在文件 `hy2410.patch` 中。`-N` 参数使得比较时把不存在的文件作为空文件看待, `-u` 参数是为了在有差异的内容前后保留若干行相同的内容以作为参考, `-r` 参数则表示比较整个目录树下的所有文件。

四、tree

`tree` 是一个用于列出目录树的工具, 可用以下命令安装:

```
sudo apt-get install tree
```

`tree` 的基本用法如下:

```
tree [-a] [-d] [-f] [-i] [-L level] [dir ...]
```

其各个参数的含义解释如下。

- ◆ `-a`: 列出所有文件 (包括隐藏文件)。
- ◆ `-d`: 只列出目录。
- ◆ `-f`: 列出文件的完整路径。
- ◆ `-i`: 去掉输出中的导引线, 只留下文件名 (经常与 `-f` 一起使用)。
- ◆ `-L level`: 只列出到第 `level` 层子目录。
- ◆ `dir`: 被输出的目录, 如果省略, 默认为当前目录。

五、ctags

`ctags` 用于生成 C 语言源文件的标签文件, 结合 `vi` 编辑器可以快速在多个源文件中查找符号的定义, 可用以下命令安装:

```
sudo apt-get install exuberant-ctags
```

安装后用如下命令可生成当前目录树下所有源文件的标签文件:

```
ctags -R
```

六、flex 和 bison

`flex` 用于自动生成词法分析器, `bison` 用于自动生成语法分析器, 很多软件的编译要用到它们, 可用如下命令安装:

```
sudo apt-get install flex bison
```

七、gettext

`gettext` 是 Linux 上的国际化编程工具, 可用以下命令安装:

```
sudo apt-get install gettext
```

八、mkfs.jffs2

`mkfs.jffs2` 在嵌入式开发中用来制作 JFFS2 格式的文件系统映像, 可用以下命令安装:

```
sudo apt-get install mtd-tools
```

九、realpath

realpath 是一个小工具，用来得到文件最终的绝对路径，可用以下命令安装：

```
sudo apt-get install realpath
```

十、strace

strace 用于跟踪应用程序的系统调用，常用于调试程序，可用以下命令安装：

```
sudo apt-get install strace
```

十一、wireless-tools

wireless-tools 是无线上网工具包，包含 **iwconfig** 和 **iwlist** 等工具软件，可用以下命令安装：

```
sudo apt-get install wireless-tools
```

十二、中文输入法

Linux 上常见的中文输入法有 **scim**、**fcitx** 等。以 **fcitx** 为例，可用以下命令安装：

```
sudo apt-get install fcitx
```

安装之后，用以下命令选择默认的输入法：

```
im-switch -c
```

选择后重新登录图形界面才生效。

fcitx 的配置文件是 `~/.fcitx/config`。由于文件中包含 GB2312 编码的汉字，因此在没有 GB2312 编码支持的系统上显示为乱码。

3.4.3.3 开发包

Debian 上有一类软件包，它们的名称往往以 **-dev** 为后缀，可称为开发包。这些包的内容一般是某个开发平台的头文件、文档和静态库等，当编译基于这个平台的程序时就需要安装开发包。

一、libncurses5-dev

curses 是一个命令行界面下模拟图形界面的共享库。如果要以菜单方式对一些软件（如内核、**busybox** 等）进行配置，就需要它的开发包，可用如下命令安装：

```
sudo apt-get install libncurses5-dev
```

二、xorg-dev

xorg 是 **Xwindow** 平台的提供者。如果要编译一些基于 **Xwindow** 的软件（如 **Qt** 等），就需要它的开发包，可用如下命令安装：

```
sudo apt-get install xorg-dev
```

3.4.4 从源码安装软件

Linux 上的很多软件以源码形式发布，有时必须从源码安装所需要的软件。从源码安装软件的一些基本步骤如下。

一、解压缩源码

源码一般是以压缩文件形式发布的，因此要先解压缩。

二、建立工作目录

编译之前一般要在源码目录之外建立一个工作目录，编译工作将在新建立的目录中执行，目的是避免编译产生的文件对源码有“污染”，这样当编译出错时可以直接将工作目录删除重来。假设源码解压缩后的目录为 **software**，则可以这样建立新目录：

```
mkdir software-build
```

也有一些软件不支持在另外的目录中编译，这样就只能直接在源码目录中编译。

三、配置

首先进入工作目录：

```
cd software-build
```

调用源码中的 **configure** 脚本进行配置：

```
../software/configure
```

configure 所支持的配置参数一般可用 **--help** 参数查询：

```
../software/configure --help
```

配置完成后工作目录中将出现 **Makefile** 等文件。

四、编译

输入 **make** 命令开始编译：

```
make
```

编译成功后，工作目录中就有了相应的可执行文件和共享库等。

五、安装

用以下命令进行安装：

```
sudo make install
```

安装过程一般要向系统目录复制文件，所以要有 **root** 权限。



因为从源码安装的软件不进入系统的软件包管理体系，所以有与系统已安装软件发生冲突的可能性。

六、卸载

有些软件源码支持用以下命令进行卸载操作：

```
sudo make uninstall
```

但很多源码并不支持。

七、交叉编译

交叉编译的步骤与上述步骤相似，一般在配置阶段设置交叉编译的参数，如：

```
../software/configure --host=arm-linux --prefix=../sysroot/usr
```

--host 参数用于指定目标平台，这样配置以后的 Makefile 会使用 arm-linux-gcc 工具链作为编译器。--prefix 用于指定安装目录。显然，交叉编译后的软件不应该装在默认的系统目录里。

3.4.5 安装编译环境

Debian 5.0 的默认安装选项不包括开发环境。我们可以在系统安装好后通过本节介绍的内容建立本地编译环境。基本的编译工具包括 gcc 工具链、make 工具等。

3.4.5.1 安装 gcc 工具链

使用以下命令可安装 Debian 上最新版本的 gcc 工具链：

```
sudo apt-get install gcc
```

以上安装的只是 C 编译器。Debian 将 GNU 的 C 编译器和 C++ 编译器分为两个软件包发布，因此还要单独安装 C++ 编译器：

```
sudo apt-get install g++
```

由于 GNU 编译器在版本升级时对汇编、C 语言及 C++ 的语法分析有一些调整，故某些早期的软件源码在新版本的编译器上可能无法编译通过，因此有时需要安装较老版本的编译器。事实上，Debian 是将差异较大的不同版本编译器各自作为独立的软件包发布的。

以安装 gcc-3.3 为例，这个软件包是在 Debian 4.0 发行版（代号 etch）中发布的，5.0 版本中已经不存在，因此需要修改软件源配置文件 /etc/apt/sources.list，在其中加入以下一行：

```
deb http://ftp.debian.org/debian etch main
```

安装方法类似：

```
sudo apt-get install gcc-3.3
sudo apt-get install g++-3.3
```

当主机上同时存在多个版本的编译器时，就有版本切换的问题。一般来说，只需要切换 cpp，gcc，g++ 三个关键工具即可，而这三个工具实际上都只是符号链接，故可以通过修改链接的方法切换到编译器的 3.3 版本：

```
sudo ln -fsv /usr/bin/cpp-3.3 /usr/bin/cpp
sudo ln -fsv /usr/bin/gcc-3.3 /usr/bin/gcc
sudo ln -fsv /usr/bin/g++-3.3 /usr/bin/g++
```

以上方法稍显麻烦，其实 Debian 提供了解决软件多版本切换问题的替换机制，利用它可以解决这一问题。首先将两个版本的 gcc 加入到一个替换组：

```
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-3.3 33
```



```
--slave /usr/bin/cpp cpp /usr/bin/cpp-3.3 --slave /usr/bin/g++ g++
/usr/bin/g++-3.3
sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.3 43
--slave /usr/bin/cpp cpp /usr/bin/cpp-4.3 --slave /usr/bin/g++ g++
/usr/bin/g++-4.3
```

以上两条命令将 `/usr/bin/gcc-3.3` 和 `/usr/bin/gcc-4.3` 设为 `gcc` 替换组的两个替换项，分别指定其优先级为 33 和 43。命令中同时还指定了 `cpp` 和 `g++` 作为从替换组，这样当 `gcc` 切换的时候，`cpp` 和 `g++` 也会同时切换。

然后用以下命令可以通过交互的方式切换版本：

```
sudo update-alternatives --config gcc
```

或者直接指定所需的替换项：

```
sudo update-alternatives --set gcc /usr/bin/gcc-4.3
```

以下命令可查看 `gcc` 替换组中的所有替换项：

```
sudo update-alternatives --list gcc
```

3.4.5.2 安装 make 工具

安装 `make` 工具使用的命令如下：

```
sudo apt-get install make
```

这些基本工具一般就在 `Debian` 的第一张光盘上，只是默认的安装选项没有包括。

3.4.5.3 安装开发手册

开发手册的安装方法如下：

```
sudo apt-get install manpages-dev
```

因版权问题，关于 `POSIX` 标准接口的手册放在另外的软件包里：

```
sudo apt-get install manpages-posix
sudo apt-get install manpages-posix-dev
```

安装开发手册后，我们就可以随时查阅 `C` 标准库中函数的使用说明了。

还可以安装中文版手册：

```
sudo apt-get install manpages-zh
```

中文版手册装在 `/usr/share/man/zh_CN` 目录中。为了让 `man` 命令能够查到中文版手册，需要修改其配置文件 `/etc/manpath.config`，加入以下命令：

```
MANDATORY_MANPATH /usr/share/man/zh_CN
```

需要指出的是，目前中文版的手册还很不完整，参考意义不大。



3.4.5.4 安装内核编译环境

如果要在主机上编译内核或内核模块，必须先安装内核头文件：

```
sudo apt-get install linux-headers-$(uname -r)
```

安装后的内核头文件放在 `/usr/src/linux-headers-$(uname -r)` 目录下，但实际上内核编译时一般将 `/lib/modules/$(uname -r)/build` 作为内核目录，这个目录是指向前者的一个符号链接。

3.4.5.5 安装 gdb

`gdb` 是程序调试工具，可用以下命令安装：

```
sudo apt-get install gdb
```

3.5 建立交叉编译环境

建立交叉编译环境可以有两种选择：一种是下载各种源码进行编译安装，这是一个烦琐且容易出错的过程；另一种是直接从网上下载已编译好的工具链进行安装。

3.5.1 下载安装

3.3.2 版本的交叉编译工具链可用以下命令下载：

```
wget http://www.handhelds.org/download/projects/toolchain/arm-linux-gcc-3.3.2.tar.bz2
```

这个压缩包里包含了从根目录开始的路径，直接解压到根目录即可使用：

```
sudo tar -C / -xjf arm-linux-gcc-3.3.2.tar.bz2
```

使用时设置环境变量：

```
export PATH=/usr/local/arm/3.3.2/bin:${PATH}
```

3.4.1 版本的交叉编译工具链可用以下命令下载：

```
wget http://www.handhelds.org/download/projects/toolchain/arm-linux-gcc-3.4.1.tar.bz2
```

安装与使用方法与 3.3.2 版本类似。

3.5.2 从源码编译安装

下面将以 `gcc-3.4.4` 和 `gcc-3.3.6` 为例说明从源码编译出交叉编译工具链的方法。需要注意的是，编译器的源码一般要用比它版本低的编译器来编译，这里用的是主机上的 `gcc-3.3` 编译器。

3.5.2.1 arm-linux-gcc-3.4.4

一般来说，基本的编译工具链包括二进制工具、编译器、内核头文件、C 标准库等组成部分，它们构成一个有机的整体，因此其编译过程也是相互关联的。

在编译过程中，我们将使用一些环境变量，可如下设置：



```
TARGET=arm-linux # 编译目标
SYSROOT=/usr/local/arm/sysroot # 目标系统根目录
CCDIR=/usr/local/arm/3.4.4 # 编译器安装目录
PATH=${CCDIR}/bin:$PATH # 修改 PATH 以使新的编译器能被找到
export TARGET SYSROOT CCDIR PATH # 导出以上环境变量
```

编译所用到的源码可用以下命令下载：

```
wget ftp://ftp.gnu.org/gnu/binutils/binutils-2.16.tar.bz2
wget ftp://ftp.gnu.org/gnu/gcc/gcc-3.4.4/gcc-3.4.4.tar.bz2
wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.10.tar.bz2
wget ftp://ftp.gnu.org/gnu/glibc/glibc-2.3.5.tar.bz2
wget ftp://ftp.gnu.org/gnu/glibc/glibc-linuxthreads-2.3.5.tar.bz2
```

直接用以上源码并不能顺利编译通过，还需要下载以下补丁：

```
wget 'http://gcc.gnu.org/cgi-bin/cvsweb.cgi/gcc/gcc/flow.c.diff?r1=1.563.4.2&r2=1.563.4.3&only_with_tag=csl-arm-branch' -O flow.c.diff
wget http://frank.harvard.edu/~coldwell/toolchain/ioperm.c.diff
wget http://frank.harvard.edu/~coldwell/toolchain/t-linux.diff
```

把以上源码和补丁都放在同一个目录下，然后就可以开始安装了。下面将分阶段介绍编译的过程。

一、安装 binutils

binutils 提供了诸如汇编器等一套二进制工具，必须首先编译：

```
tar -xjvf binutils-2.16.tar.bz2
mkdir binutils-2.16-build
cd binutils-2.16-build
../binutils-2.16/configure --prefix=${CCDIR} --target=${TARGET} --with-sysroot=${SYSROOT}
make
sudo make install
cd ..
```

二、安装内核头文件

安装步骤如下：

```
tar -xjf linux-2.6.10.tar.bz2
cd linux-2.6.10
make s3c2410_defconfig ARCH=arm
make include/linux/version.h ARCH=arm
sudo mkdir -p ${SYSROOT}/usr/include
sudo cp -r include/asm-arm ${SYSROOT}/usr/include/asm
sudo cp -r include/asm-generic ${SYSROOT}/usr/include
sudo cp -r include/linux ${SYSROOT}/usr/include
cd ..
```



三、安装 glibc 头文件

glibc 提供 C 标准库，它需要编译成目标平台的共享库。由于交叉编译器还没有生成，故无法完整编译。但编译 gcc 却需要 glibc 的头文件，需先行安装，步骤如下：

```
tar -xjf glibc-2.3.5.tar.bz2
patch -d glibc-2.3.5 -p1 < ioperm.c.diff
tar -C glibc-2.3.5 -xjvf glibc-linuxthreads-2.3.5.tar.bz2
mkdir glibc-2.3.5-build
cd glibc-2.3.5-build
../glibc-2.3.5/configure --prefix=/usr --host=${TARGET} --enable-add-ons=
linuxthreads --with-headers=${SYSROOT}/usr/include
sudo make install-headers cross-compiling=yes install_root=${SYSROOT}
sudo touch ${SYSROOT}/usr/include/gnu/stubs.h
sudo touch ${SYSROOT}/usr/include/bits/stdio_lim.h
cd ..
```

四、安装“简版”工具链

gcc 和 g++ 在同一个源码包中，第一次编译时只能得到 gcc，因为编译 g++ 需要目标平台的 glibc 库支持。编译步骤如下：

```
tar -xjf gcc-3.4.4.tar.bz2
patch -d gcc-3.4.4 -p1 < flow.c.diff
patch -d gcc-3.4.4 -p1 < t-linux.diff
mkdir gcc-3.4.4-build
cd gcc-3.4.4-build
../gcc-3.4.4/configure --prefix=${CCDIR} --target=${TARGET} --enable-languages=
c --with-sysroot=${SYSROOT}
make all-gcc
sudo PATH=${PATH} make install-gcc
cd ..
```

五、安装 glibc

有了交叉编译器 arm-linux-gcc 后，就可以安装 glibc 了，过程如下：

```
cd glibc-2.3.5-build
sudo rm -rf *
../glibc-2.3.5/configure CC=arm-linux-gcc --prefix=/usr --build=i686-linux
--host=${TARGET} --enable-add-ons=linuxthreads --with-headers=${SYSROOT}/
usr/include --without-__thread
make
sudo PATH=${PATH} make install install_root=${SYSROOT}
cd ..
```



安装时 install_root 的设置特别重要，如果不设置，则目标平台（例子中为 ARM 平台）的 C 标准库会直接将主机（x86 平台）的 C 标准库覆盖，系统马上瘫痪。这种错误很难恢复。

六、安装“全版”工具链

此时可以重新编译工具链，将 g++ 也编译出来：



```
cd gcc-3.4.4-build
rm -rf *
../gcc-3.4.4/configure --prefix=${CCDIR} --target=${TARGET} --enable-languages=
c,c++ --with-sysroot=${SYSROOT}
make
sudo PATH=${PATH} make install
cd ..
```

至此，一个基本的交叉编译工具链就成形了。

3.5.2.2 arm-linux-gcc-3.3.6

在已经有一个交叉编译工具链的基础上，可以直接把另外一个版本的工具链编译出来。以 gcc-3.3.6 为例，源码可用以下命令下载：

```
wget ftp://ftp.gnu.org/gnu/gcc/gcc-3.3.6/gcc-3.3.6.tar.bz2
```

环境变量要进行修改，以便将这个版本的编译工具链装在另外一个目录：

```
TARGET=arm-linux # 编译目标
SYSROOT=/usr/local/arm/sysroot # 目标系统根目录
CCDIR=/usr/local/arm/3.3.6 # 编译器安装目录
PATH=${CCDIR}/bin:$PATH # 修改 PATH 以使新的编译器能被找到
```

首先 binutils-2.16 要重新安装到新的目录，安装的过程与安装 gcc-3.4.4 相同，不过这时 CCDIR 环境变量已经与原来不同。然后可以直接编译“全版”工具链：

```
tar -xjf gcc-3.3.6.tar.bz2
mkdir gcc-3.3.6-build
cd gcc-3.3.6-build
../gcc-3.3.6/configure --prefix=${CCDIR} --target=${TARGET} --enable-languages=
c,c++ --with-sysroot=${SYSROOT}
make
sudo PATH=${PATH} make install
cd ..
```

注意观察我们在两个版本的 gcc 源码配置时使用了参数 `--with-sysroot=${SYSROOT}`，而 glibc 也安装到了这一目录，因此实际上它们用的是同一套 C 标准库。这种做法有一个好处：`${SYSROOT}` 这个目录添加一些基本的系统目录和配置，再安装 busybox，就可以直接作为目标机的根文件系统。

因为两个版本的交叉编译工具链分别装在两个目录下，所以只需修改 PATH 环境变量就可以切换所使用的编译器版本。

3.5.2.3 arm-linux-gdb

在调试目标机时，通常采用远程调试的方式，在目标机上运行 gdbserver，主机上用 arm-linux-gdb 通过网络与目标机连接，对目标机上运行的程序进行控制和观察。

可以从 gdb 的源码编译得到 arm-linux-gdb。首先下载源码：



```
wget ftp://ftp.gnu.org/gnu/gdb/gdb-6.8.tar.bz2
```

一般情况下, `gdb` 都与编译工具链放在一起。安装过程如下:

```
tar -xjf gdb-6.8.tar.bz2
mkdir gdb-6.8-build
cd gdb-6.8-build
../gdb-6.8/configure --target=${TARGET} --prefix=${CCDIR}
make
sudo make install
```

然后编译 `gdbserver`, 放在目标机的根文件系统上:

```
mkdir gdb/gdbserver
cd gdb/gdbserver
../../../../gdb-6.8/gdb/gdbserver/configure --host=${TARGET} --prefix=${SYSROOT}/usr
make
arm-linux-strip gdbserver
sudo make install
cd ../../../../
```

`arm-linux-strip` 的作用是去除可执行文件中的符号表, 使之更小。

3.6 vi 编辑器

`vi` 编辑器是 Linux 系统上常用的文本编辑器, 它的改进版本 `vim` 增加了很多扩展功能, 使之更适合编辑各种程序语言的源代码。

3.6.1 vi 的工作模式

`vi` 与其他编辑器的很大不同在于它有两种基本工作模式: 命令模式和编辑模式。在命令模式下, 输入的字符作为命令使用, 不进入被编辑的文档中; 在编辑模式下, 输入的字符将修改文档的内容。在命令模式下, 如果输入某些编辑命令, 则切换到编辑模式; 在编辑模式下, 按 `Esc` 键可切换回命令模式。如果仔细划分, 则命令模式下还有普通模式、命令行模式和选择模式的区别, 编辑模式还可以分为插入模式和替换模式。

在普通模式下, 命令输入完毕就执行, 不需要回车确认, 执行完毕后还在普通模式下, 输入一个冒号则进入命令行模式; 在命令行模式下, 输入命令后需回车确认才执行, 执行完毕后返回普通模式。

在普通模式下输入命令之前, 可以先输一个数字作为命令重复执行的次数; 在命令行模式下则往往可以在前面加一个范围限定, 表示只对文档的指定部分进行操作。



`vim` 是一种行编辑器, 每一行都会以回车结束。很多 `vi` 命令都是针对文本行来设计的。



3.6.2 普通模式

普通模式下的常用命令可大致分为移动光标类、滚动屏幕类、编辑类等，如表 3.5 所示。

表 3.5 普通模式常用命令

移动光标类	
h	向左移动一个字符
l	向右移动一个字符
k	向上移动一行
j	向下移动一行
0	移动到行首
\$	移动到行尾
w	向后移动一个单词
b	向前移动一个单词
gg	移动到文档开头，如果命令前先输一个数字，则移动到指定行
G	移动到文档末尾，如果命令前先输一个数字，则移动到指定行
n	有搜索内容时移动到后一个匹配处
N	有搜索内容时移动到前一个匹配处
滚动屏幕类（同时光标也在文档中移动）	
Ctrl+F	向下滚动一屏
Ctrl+B	向上滚动一屏
Ctrl+U	向上滚动半屏
Ctrl+D	向下滚动半屏
编辑类	
x	删除光标处的字符，删除内容进入寄存器
X	删除光标处字符的前一个字符，删除内容进入寄存器
i	切换到插入模式，插入位置在光标处的字符之前
a	切换到插入模式，插入位置在光标处的字符之后
I	切换到插入模式，插入位置在光标所在行之首
A	切换到插入模式，插入位置在光标所在行之尾
o	切换到插入模式，插入位置在光标所在行之下（插入新的一行）
O	切换到插入模式，插入位置在光标所在行之上（插入新的一行）
J	将光标所在行与下一行连接成一行（删除两行间的回车）
d	紧接一个移动光标的命令，可删除相应范围内的文档内容，删除内容进入寄存器
dd	删除光标所在行，删除内容进入寄存器
D	从光标所在字符删除到行尾，删除内容进入寄存器
r	用随后输入的字符替换光标处的字符
R	进入替换模式，输入内容替换原内容
u	撤销最近的编辑操作



(续表)

复制与粘贴类	
y	紧接一个移动光标的命令，将相应范围内的文档内容复制到寄存器
p	将寄存器中的内容粘贴在光标所在处后面，如果是整行，粘贴在所在行后面
P	将寄存器中的内容粘贴在光标所在处前面，如果是整行，粘贴在所在行前面
文件类	
ZZ	存盘退出

在普通模式下输入小写字母 **v** 进入选择模式，输入大小字母 **V** 进入按行选择模式，按 **Ctrl+V** 组合键进入按列选择模式。在选择模式下，移动光标类命令兼有改变选择区域的作用，编辑类和复制粘贴类命令则直接对所选区域产生作用并返回普通模式。

3.6.3 命令行模式

命令行模式下的常用命令如表 3.6 所示。

表 3.6 命令行模式常用命令

命令	说明
:help	查看帮助首页
:0	移动到第一行
:\$	移动到最后一行
:3	移动到第 3 行
:w	保存文件
:q	退出文件
:n	新建文件
:wq	保存并退出

在执行文件保存或退出命令时可能会失败，如保存时发现文件是只读的，或者退出时发现新的修改尚未保存，这时可以在命令后加一个惊叹号 **!** 表示强制执行。

命令 **/** 和 **?** 用于搜索匹配指定模式的字符串，如：

```
/Linux
```

将从光标所在处向后搜索字符串 **Linux**。**?** 命令则是从光标所在处向前搜索。这两个命令可以直接输入，不需要先输入一个冒号。执行一次搜索命令后，要搜索的模式记在寄存器 **/** 中，以后可以直接用普通模式命令 **n** 或者 **N** 进行下一次搜索。

替换命令是 **s**，用法举例如下：

```
:%s/Linux/linux/g
```

这条命令将把文档中所有的 **Linux** 替换成 **linux**。其中 **%** 表示在全文范围内搜索，最后的 **g** 表示如果同一行内有多匹配，将全部替换，否则只替换第一个。

3.6.4 寄存器

vim 的寄存器类似于其他操作系统上的剪贴板，被删除或复制的内容进入寄存器，然后可以用粘贴命令插入到文档中。**vim** 同时有多个寄存器，可以保存不同的内容，其中部分寄存器有特殊用途，因此一般只使用以字母从 **a** 到 **z** 命名的 26 个寄存器。

与寄存器有关的普通模式命令都支持选择寄存器，方法是：输入命令之前先输入双引号 " 加寄存器的名称。如以下命令可将整个文档的内容复制入 **a** 寄存器：

```
gg"ayG
```

使用这些寄存器时，如果将小写字母换成对应的大写字母，则表示新进入的内容要附加在原内容的后面。

这些用字母命名的寄存器仅限于 **vim** 内部使用，而寄存器 ***** 和寄存器 **+** 则与操作系统的剪贴板是相关的。

3.6.5 与编程有关的技巧

在普通模式下可以使用以下命令：如果光标处是括号，则使用 **%** 命令可以移动光标到匹配的括号处；如果光标处是变量名，则使用 **gd** 命令可以移动光标到局部变量的定义处；如果光标处是变量名，则使用 **gD** 命令可以移动光标到全局变量的定义处；如果光标处是一个系统函数的名称，则按 **Ctrl+K** 组合键将调用 **man** 命令查询它的手册页。

在命令行模式下先输入一个惊叹号 **!** 再输入 **Shell** 命令可执行此命令，当使用 **make** 命令时可以省略 **!** 符号。

在 **vim** 的配置文件 **/etc/vim/vimrc** 中增加如下一句可开启 **vim** 的自动缩进功能：

```
filetype plugin indent on
```

在普通模式下用命令 **=** 可以对代码进行自动重新排版，用 **==** 命令可对光标所在行重排，用 **=** 命令加一个光标移动的命令可对相应范围内的行重排，选择模式下用 **=** 命令则对所选区域进行重排。例如，用以下命令对全部代码进行重排：

```
gg=G
```

用 **ctags** 工具为源码生成 **tags** 文件之后，在 **vim** 中就可以根据 **tags** 文件中的信息，从符号（类型名、变量名、函数名、宏名等）的使用处跳转到定义处，方法是：当光标在符号上时按 **Ctrl+]** 组合键。这种跳转可以跨越文件，如果想回到光标原来的位置，可按 **Ctrl+T** 组合键。

3.7 gcc 工具链

GNU/Linux 系统上常用的编译工具是 **gcc**。**gcc** 实质上不是一个单独的程序，而是多个程序的集合，因此称为工具链。

3.7.1 编译过程

从 C 语言源码到可执行程序一般要经过以下的处理步骤。

一、预处理

在这一阶段，源码中的所有预处理语句得到处理，例如：

- ◆ `#include` 语句所包含的文件内容替换掉语句本身。
- ◆ 所有已定义的宏被展开。
- ◆ 根据 `#ifdef`，`#if` 等语句的条件是否成立取舍相应的部分。

预处理之后源码中不再包含任何预处理语句。`gcc` 工具链中进行预处理的工具为 `cpp`。

二、编译

在预处理过的源码基础上开始编译过程。这一阶段中，编译器对源码进行词法分析、语法分析、优化等操作，最后生成汇编代码。这是整个过程中最重要的一步，因此也常把整个过程称为编译。`gcc` 工具链中进行编译的工具为 `gcc`。

三、汇编

这一阶段使用汇编器对汇编代码进行处理，生成机器语言代码，保存在目标文件中。目标文件中除了有机器代码，还保存一些其他信息，如符号表、重定位表等。`gcc` 工具链中的汇编器为 `as`。

四、链接

经过汇编以后的机器代码还不能直接运行。为了使操作系统能够正确加载可执行文件，文件中必须包含固定格式的信息头，还必须与系统提供的启动代码链接起来才能正常启动，这些工作都是由链接器来完成的。链接器能够将多个目标文件衔接在一起，进行重定位，最后生成可执行文件（或共享库）。`gcc` 工具链中的链接器为 `ld`。

整个编译的过程可用图 3.4 来说明，其中虚线框表示的文件大多数情况下只是临时文件，甚至根本不作为文件保存。

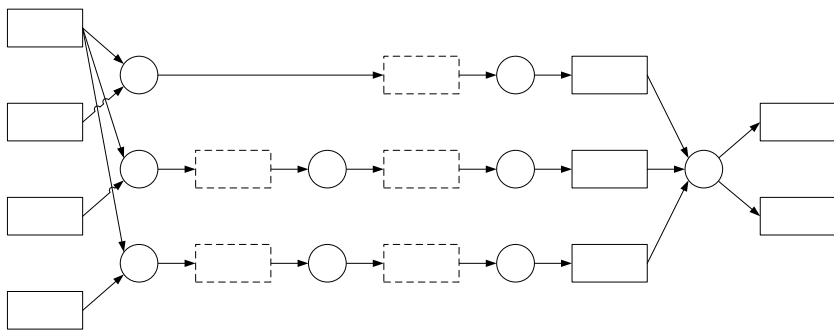


图 3.4 从源码到可执行代码的编译流程

3.7.2 gcc 用法

虽然编译过程的每一步是由不同的工具完成的，但使用时可以只通过 `gcc` 这个工具，它会根据需要调用其他工具来完成相应的功能。

3.7.2.1 基本用法

`gcc` 的基本用法如下：

```
gcc [-pipe] [-Wall] [-O1..3] [-g] [-o name] file ...
```

其各个参数的含义解释如下。

- ◆ `-pipe`: 使用管道而不是临时文件传递中间结果, 能加快编译过程。
- ◆ `-Wall`: 打开所有警告项。
- ◆ `-O`: 设置优化级别, `O0` 表示关闭优化功能。
- ◆ `-g`: 将调试信息编译到目标文件中。
- ◆ `-o name`: 指定输出文件的名称是 `name`。
- ◆ `file`: 被编译 (链接) 的文件。

使用举例:

```
gcc myprog.c # 单个 C 源文件编译链接出可执行文件, 文件名默认为 a.out
gcc a.c b.c -o app # 两个 C 源文件编译链接出可执行文件 app
gcc a.S b.c -o app # 汇编和 C 源文件编译链接出可执行文件 app
gcc a.o b.o -o app # 两个目标文件链接出可执行文件 app
```

3.7.2.2 分阶段编译

很多时候并不需要将整个编译过程进行到底, 只需得到中间结果即可, 这可以通过使用 `gcc` 的某些参数来做到, 例如:

```
gcc -E a.c # -E 参数使编译进行到预处理后停下, 结果在标准输出上
gcc -S a.c # -S 参数使编译进行到汇编文件时停止, 生成文件 a.s
gcc -c a.c # -c 参数使编译进行到目标文件时停止, 生成文件 a.o
```

对于有多个源文件的大型 C 程序, 通常会分两个阶段进行编译: 首先使用 `-c` 参数将各个源文件分别编译成目标文件, 然后将所有目标文件链接起来。这样做的好处是当只有一个源文件发生改变时, 可以只编译这个源文件, 然后重新链接即可, 不必全部重新编译。

3.7.2.3 定义宏

在 `gcc` 的参数上可以定义宏, 所定义的宏等价于在源文件中直接定义, 如:

```
gcc -DDEBUG=yes main.c # -D 参数用于定义宏
```

等价于 `main.c` 的开头有如下一行:

```
#define DEBUG yes
```

3.7.2.4 头文件和共享库

`gcc` 在进行编译预处理时, 要在工具链默认的一些头文件目录中搜索需要的头文件, 这些目录一般包括 `/usr/local/include` 和 `/usr/include`。如果希望 `gcc` 能够到其他目录中查找头文件, 则需要在参数上指定, 如:

```
gcc -I~/include main.c # -I 参数用于增加头文件的查找目录
```

如果源码中使用了共享库中的函数, 则目标文件就要与相应的共享库链接起来。`gcc` 默认会链

接 C 共享库，因为它提供了基本的系统调用接口，几乎所有的程序都会用到它。如果要链接其他共享库则必须在参数上指定，例如：

```
gcc -lpthread main.c # -l 参数用于指定要链接的共享库
```

pthread 是线程库，如果程序中用了线程的相关函数，必须与这个库链接。要注意，共享库的文件名一般是 lib*.so，但在 -l 参数后面时不需要写全，编译器会自动将名字补全成文件名。

链接共享库时 gcc 到默认的一些目录中去查找，这些目录一般包括 /usr/lib 和 /lib。如果要链接的共享库在其他目录中，则需要用 -L 参数指定，如：

```
gcc -L~/src/sqlite3 -lsqlite3 main.c # -L 参数指定查找共享库的目录
```

3.7.2.5 编译 C++ 源码

编译 C++ 源码时要用 g++ 工具，其基本用法与 gcc 相同，不再赘述。

3.7.2.6 交叉编译

交叉编译工具链中的工具名称是在本地编译工具的相应名称前加上一个统一的前缀，如对应 gcc 的 ARM 工具是 arm-linux-gcc，对应 cpp 的 ARM 工具是 arm-linux-cpp。它们的用法基本一致，但需要注意的是，它们默认的头文件搜索路径和共享库搜索路径与本地编译器不同，由交叉编译工具链时的一些配置参数决定。

3.8 make 与 Makefile

如果一个软件包含成百上千个源文件，那么从源文件编译链接出最终的可执行文件的过程将烦琐不堪。这时可以使用 make 工具对编译过程进行管理，它会根据一个文件中的内容自动执行命令，这个文件默认是当前目录下的 Makefile 或 makefile，一般也直接称这个文件为 Makefile 而不管它的名称是什么。

make 工具有目标管理的功能，在 Makefile 中可以定义多个目标并给出它们之间的依赖关系。为了完成一个目标，make 工具会先完成它所依赖的目标。如果目标是文件，make 还会检查它们的时间戳，如果一个目标比它所有的依赖目标都新，那么就不会执行生成这个目标的命令，这样就节约了编译过程所消耗的时间。

3.8.1 make 工具的使用

make 工具的基本用法如下：

```
make [-C dir] [-f file] [target ...]
```

其各个参数的含义解释如下。

- ◆ -C dir: 执行时进入 dir 目录，默认是当前目录。
- ◆ -f file: 使用 file 作为 Makefile。
- ◆ target: 要完成的目标，目标在 Makefile 中定义，默认是定义的第一个目标。



3.8.2 Makefile

3.8.2.1 目标与生成规则

要使用 `make` 工具，首先要撰写 `Makefile`。在 `Makefile` 中定义目标的格式如下：

```
target: dep1 dep2 # dep1 dep2 是 target 依赖的目标
```

其中 `target` 是所定义的目标，它可以是一个真实的文件名，也可以是伪目标。

定义目标之后可以给出它的生成规则，实际上就是一条或多条 `Shell` 命令，这些命令前面必须加一个 `Tab` 键缩进。当然，`make` 命令本身也可以出现在生成规则里，这时要小心产生循环嵌套。



如果 `Makefile` 中非生成规则的内容前加了 `Tab` 键，也会被误认为生成规则。

在执行生成规则中的命令时，如果发生错误，则 `make` 命令立刻退出。如果某条命令的出错无关紧要，则可以在这条命令的前面加一个减号 `-`，这样即使命令出错，`make` 也会继续执行剩余的命令。

以下是一个简单的 `Makefile` 的例子：

```
TARGET := test # 定义变量 TARGET 保存可执行文件名
SRCS := a.c b.c # 定义变量 SRCS 保存源文件列表
CFLAGS += -pipe -Wall -O2 # 增加 gcc 的编译参数
OBJS := $(SRCS:.c=.o) # 把 SRCS 变量中的 .c 替换成 .o 构造出目标文件列表
all: $(TARGET) # all 目标，依赖于要得到的可执行文件
$(TARGET): $(OBJS) # 可执行文件依赖于所有目标文件
        $(CC) -o $@ $^ # 可执行文件的生成规则
### 由 .c 生成 .o 的规则已内置在 GNU make 中，可省略 ###
clean: # clean 目标，用于清理所有编译产生的文件
        -rm -f $(OBJS) $(TARGET)
```

3.8.2.2 变量

`Makefile` 中也可以定义变量，正确使用变量有助于 `Makefile` 的维护，例如在上述的 `Makefile` 中，只需要修改 `SRCS` 变量的值即可适用于其他源码。

定义或修改变量值的方法是使用赋值操作，如表 3.7 所示。

表 3.7 `Makefile` 中的赋值操作

符号	功能
<code>:=</code>	直接赋值，新值覆盖原来的值
<code>?=</code>	条件赋值，如果原来无值则赋值，否则保持原来的值
<code>+=</code>	加法赋值，新值附加在原来的值后面
<code>=</code>	递归赋值，如果右侧包含其他变量，当这些变量的值变化时，被赋值变量的值也随之变化

在 `Makefile` 中定义变量时，赋值符号两边可以有空格。注意在 `Makefile` 中使用变量时用的



是 `$()` 而不是 Shell 中的 `${}`，如果变量名只是一个字符，则小括号可以省略。

Makefile 中也支持类似于 Shell 中的命令替换，例如：

```
PWD := $(shell pwd) # $(shell pwd) 将被执行 Shell 命令 pwd 的结果替换
```

其中 `shell` 关键字不可省略。事实上，`make` 在执行时会自动将 Shell 中的环境变量导入，如同在 Makefile 中定义的变量一样，因此上述语句即使不写，`PWD` 变量也能得到应有的值。

Makefile 中还可以使用几个特殊变量，如表 3.8 所示。

表 3.8 Makefile 中的特殊变量

变量	含义
<code>\$@</code>	用在生成规则中，表示当前目标
<code>\$<</code>	用在生成规则中，表示当前目标的第一个依赖目标
<code>\$^</code>	用在生成规则中，表示当前目标的所有依赖目标

3.8.2.3 隐含规则

为了方便，`make` 工具内置了一些生成规则，这些规则即使不在 Makefile 中写出也可以起作用，比如从 `.c` 文件生成 `.o` 文件的隐含规则如下：

```
%.o: %.c
    $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c $< -o $@
```

其中引用了变量 `CFLAGS`，所以即使不写出这条生成规则，也可以通过这个变量改变编译时的参数。变量 `CC` 指的是 C 编译器，如果 Shell 中没有定义，则默认值为 `cc`，`cc` 往往又是一个最终指向 `gcc` 的符号链接。

如果将上述代码明确写在 Makefile 中，则可以覆盖原来的隐含规则。

3.8.2.4 伪目标

有些定义的目标并不是一个要生成的文件，只是为了完成一些操作，如常用的默认目标 `all`、用来清理源码的目标 `clean`、用来安装的目标 `install` 等。如果正好有文件的名字跟这些目标重复，则 `make` 工具就会根据时间戳决定相应的生成规则要不要执行，这与要求是不符的。为了避免这种情况，可以明确声明这些目标为伪目标，如：

```
.PHONY: all clean # 声明 all clean 为伪目标
```

3.8.2.5 分支结构

Makefile 中还支持分支结构，基本语法如下：

```
ifdef VAR # 判断 VAR 是否有定义
    list1 # 判断成立则此处内容生效，这里可以包含目标定义和生成规则
else
    list2 # 判断不成立则此处内容生效，这里可以包含目标定义和生成规则
endif
```

其中的条件判断还有以下几种形式：



```
ifndef VAR # 判断 VAR 是否未定义, 与 ifdef 相反
ifeq (STR1, STR2) # 判断 STR1 和 STR2 是否相等
ifneq (STR1, STR2) # 判断 STR1 和 STR2 是否不相等, 与 ifeq 相反
```

3.8.2.6 自动生成依赖关系

目标文件 `.o` 的隐含依赖关系中只包括了相应的 `.c` 源文件, 实际上应该将此源文件所包含的头文件也包括进去才合理。gcc 可以自动生成这种依赖关系, 用法如下:

```
gcc -M *.c > .depfile
```

这条命令将生成的依赖关系重定向到文件 `.depfile` 中, 其中 `-M` 参数表示要输出依赖关系, 如果不希望结果中出现系统头文件, 则将 `-M` 换成 `-MM` 即可。为了使用方便, 可在 `Makefile` 中加入以下几行:

```
dep:
    $(CC) $(CFLAGS) -MM $(SRCS) > .depfile
ifeq (.depfile, $(wildcard .depfile))
    include .depfile # include 可以包含其他文件的内容, 这里不能缩进一个 Tab 键
endif
```

这样输入 `make dep` 即可生成依赖关系, 保存在文件 `.depfile` 中。`Makefile` 中又通过 `include` 语句包含了此文件的内容, 因此依赖关系得以生效。

`$(wildcard)` 是 `Makefile` 内的函数, 用于展开通配符。当匹配的文件不存在时, 展开成空字符串。在包含文件之前进行了一次判断, 以免当文件不存在时发生错误。

这里需要明确一点, 一个目标的生成规则只能定义一次, 如果有多次定义, 则后面的规则将覆盖前面的规则, 并且 `make` 会给出警告; 依赖关系却可以多次定义, 所有的定义将被合并考虑。

3.9 gdb 调试工具

`gdb` 是 GNU 的调试工具, 它可以跟踪被调试的程序, 进行设置断点、单步执行等操作。当程序暂停执行时, 可以使用命令查看程序中的变量值、CPU 的寄存器值、内存的值以及函数调用栈等信息, 是一个功能非常强大的调试工具。

3.9.1 调试本地程序

`gdb` 工具可以对主机上的应用程序进行调试。被调试的应用程序在编译时最好使用 `-g` 参数将调试信息编入目标文件中, 如:

```
gcc -g app.c -o app
```

用以下命令可以启动对程序 `app` 的调试:

```
gdb app
```

如果程序 `app` 运行时需要参数, 则用以下命令:



`gdb --args app arg1 arg2 #` `arg1` 和 `arg2` 被视为 `app` 的参数，而不是 `gdb` 的参数

启动后进入 `gdb` 的交互界面，可以输入 `gdb` 的命令开始调试，其常用命令如表 3.9 所示。

表 3.9 `gdb` 的常用命令

命令	简写	功能
<code>list</code>	<code>l</code>	列出源码
<code>break</code>	<code>b</code>	设置断点
<code>run</code>	<code>r</code>	从头开始运行程序
<code>continue</code>	<code>c</code>	从停止处继续运行程序
<code>next</code>	<code>n</code>	向前执行一句（不进入被调用函数中）
<code>step</code>	<code>s</code>	向前执行一句（可进入被调用函数中）
<code>return</code>	<code>ret</code>	从当前函数返回
<code>print</code>	<code>p</code>	显示变量或表达式的值
<code>x</code>	<code>x</code>	显示内存值
<code>backtrace</code>	<code>bt</code>	显示调用栈
<code>quit</code>	<code>q</code>	退出 <code>gdb</code>
<code>symbol-file</code>	<code>sy</code>	从可执行文件中加载符号表

通常，直接回车就是重复上一条命令。

随 `gdb` 调试器一般还提供另外一个工具 `gdbtui`，它是一个终端上的图形界面调试器，功能及用法与 `gdb` 完全相同，只是在上方开辟了一个窗格用于显示源代码及运行情况，使用起来更直观。如图 3.5 所示是 `gdbtui` 的调试界面。

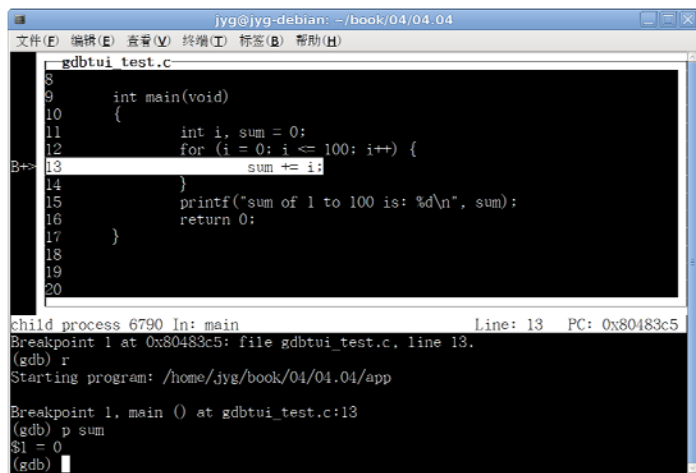


图 3.5 `gdbtui` 的调试界面

3.9.2 远程调试

调试 ARM 目标机上的应用程序时，需要用 `arm-linux-gdb` 工具。调试的方法是：在目标机上启动 `gdbserver`，在主机上启动 `arm-linux-gdb`，通过网络连接交换数据，以实现对目标机上运

行的程序进行监控。同样，交叉编译时要使用 `-g` 参数把调试信息编译进去，如：

```
arm-linux-gcc -g app.c -o app
```

如果用 TCP 连接进行调试，则在目标机上用如下命令启动 `gdbserver`：

```
gdbserver :2345 app & # 启动 gdbserver，加载程序 app 并在 2345 端口上监听
```

监听用的端口号是任意的，只要不与其他程序冲突即可。



`gdbserver` 最好是在后台运行，否则必须在另外的控制台上结束它，而目标机往往只打开一个控制台。

在主机上启动 `arm-linux-gdb`：

```
arm-linux-gdb app # 启动 arm-linux-gdb，加载程序 app
```

这里的 `app` 是目标机上的 `app` 的一个副本，它不在主机上运行，只是利用它的调试信息。在 `arm-linux-gdb` 的界面中输入以下命令开始远程调试：

```
target remote 192.168.1.128:2345
```

其中 `192.168.1.128` 是目标机的 IP 地址，`2345` 是 `gdbserver` 监听端口。此命令可简写为：

```
tar r 192.168.1.23:2345
```

连接成功后就可以像调试本地程序一样进行调试了。需要注意的是，如果被调试的程序终止，则 `gdbserver` 随之退出。因此 `run` 命令不能使用，它会先结束正在调试的程序，而启动 `gdbserver` 之后，被调试的程序其实已经在运行了。



`arm-linux-gdb` 也有一个终端图形界面的版本 `arm-linux-gdbtui` 可以使用。

3.10 buildroot 开发工具

`buildroot` 工具实际上是一些 `Makefile` 和补丁的集合，它能够根据需要自动从网络下载源码，然后执行解压缩、打补丁、配置、编译等操作，以得到嵌入式开发中常用的一些软件。这些操作只需要先进行一些简单的配置，再输入一个 `make` 即可完成，使用十分方便。一般来说，可以利用 `buildroot` 得到以下内容。

- ◆ 基于 `uClibc` 的交叉编译工具链。
- ◆ 基于 `uClibc` 编译的其他各种应用程序。
- ◆ 包含 `uClibc` 共享库的根文件系统。

`uClibc` 是一个专为嵌入式系统量身定做的轻量级 C 共享库。



buildroot 也可以编译 u-boot 映像和内核，但一般情况下，它们都要根据不同的目标机平台进行相应的修改，故通常都单独编译。

截至笔者撰稿时，最新的 buildroot 可用以下命令下载：

```
wget http://buildroot.uclibc.org/downloads/buildroot-2009.05-rc3.tar.bz2
```

下载以后首先解压源码包：

```
tar -xjf buildroot-2009.05-rc3.tar.bz2
```

进入源码目录：

```
cd buildroot-2009.05-rc3
```

采用菜单方式进行配置：

```
make menuconfig
```

buildroot 的配置界面如图 3.6 所示。

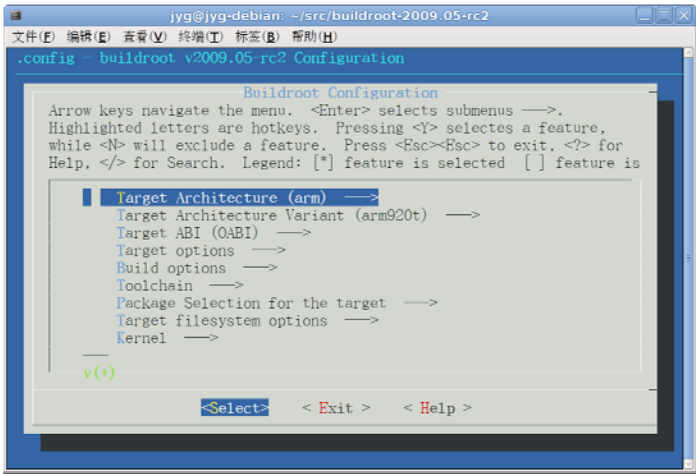


图 3.6 buildroot 配置界面

一些配置选项的说明如表 3.10 所示。

表 3.10 buildroot 配置选项

选项	说明
Target Architecture	目标平台架构
Target Architecture Variant	目标平台架构下的子类型
Target ABI	系统调用方式，应与编译内核时的选项一致
Target options	目标机的配置选项
Project name	项目名称，用于给编译过程中产生的目录命名
hostname	目标机的主机名，将写入根文件系统的 /etc/hostname

(续表)

选项	说明
Build options	编译选项
Download dir	编译过程中下载的源码所在目录，要写绝对路径
Toolchain and header file location?	编译好的工具链所在目录
Toolchain	工具链选项
Kernel Headers	内核头文件版本
uClibc C library Version	uClibc 共享库版本
Binutils Version	二进制工具 binutils 的版本
GCC compiler Version	gcc 的版本
Package Selection for the target	目标机应用程序的配置，至少应有一个 busybox
BusyBox Version	busybox 的版本

对于 HY2410A 开发板来说，它采用的 CPU 型号为 S3C2410，故目标平台架构为 ARM，子型号为 ARM920T，其他选项均可默认，退出并保存配置，然后开始编译：

```
make
```

编译完成后，在所配置的工具链目录中的 `usr/bin` 目录下可以找到 4.3.3 版本的 ARM 交叉编译工具链；在 `buildroot` 源码目录中的 `project_build_arm/uclibc/root` 是目标机的根文件系统，其中 `uclibc` 就是以所配置的项目名称命名的目录，这里用的是默认值。



Part

第 2 部分 ARM 架构与编程

第 4 章 ARM 处理器架构与编程模型

第 5 章 搭建嵌入式固件开发平台

第 6 章 S3C2410 接口与编程

第 7 章 U-boot 源码分析与移植

2

第 4 章 ARM 处理器架构与编程模型

ARM 架构是一个 32 位 RISC (Reduced Instruction Set Computing, 精简指令集) 处理器架构, 广泛地使用在许多嵌入式系统中。由于其节能的特点, ARM 架构处理器非常适用于移动通信领域。

ARM 架构有以下的 RISC 处理器特性。

- ◆ 读取/储存架构。
- ◆ 不支持地址不对齐的内存存取 (ARMv6 内核已支持)。
- ◆ 指令可以用任意的寻址方式存取数据。
- ◆ 大量的 32 位寄存器阵列。
- ◆ 固定的 32 位指令长度, 减轻解码和流水线化的负担。
- ◆ 指令的执行周期大多数均为一个 CPU 周期。

除此之外, ARM 还有如下一些特别的设计。

- ◆ 大部分指令可以条件式地执行, 降低在分支时产生的开销, 提高执行效率。
- ◆ 算术指令只会在要求时更改条件编码 (condition code)。
- ◆ 32 位筒型移位器 (barrel shifter) 可用来执行大部分的算术指令和寻址计算而不会损失效率。
- ◆ 强大的索引寻址模式 (addressing mode)。
- ◆ 精简而快速的双优先级中断子系统, 具有可切换的寄存器组。

上述特性使得 ARM 处理器在高性能、小代码尺寸、低功耗与小硅片面积间取得了一个很好的平衡。

ARM 处理器还有一些在其他 RISC 架构中所不常见的特色, 例如 PC 相对寻址 (在 ARM 上 PC 为 16 个寄存器中的一个) 以及前递加或后递加的寻址模式。

ARM7 和大多数较早的设计具有三阶段的流水线: 提取指令、解码与执行。ARM9 则有五阶段的流水线, 包含一个较快的加法器和更广的分支预测逻辑线路。这个架构使用“协处理器”来扩展指令集, 可通过指令对协处理器寻址。逻辑上通常分成 16 个协处理器, 编号分别从 0 至 15, 第 15 号协处理器 (CP15) 被保留用于某些常用的控制功能, 一般称为系统控制器。

在 ARM 架构的机器中, 外围设备连接处理器的方式通常是内存映射方式, 即把设备的寄存器映射到 ARM 的内存空间。设备也可以映射到协处理器空间, 或是连接到另外的总线控制器。

本章的主要内容是对 ARM 处理器硬件、汇编指令以及 GNU 汇编语法进行介绍, 并探讨 C 语言与汇编语言的关系。

4.1 嵌入式硬件系统

要理解嵌入式系统，首先要搞清楚以下几个问题。

一、处理器的地址映射

计算机的主要资源是存储器资源，要访问这些资源，必须先确定这些资源的地址，也就是必须将这些资源映射到处理器的地址空间（现在处理器大多数是 32 位的，就是说处理器的地址空间有 2 的 32 次方，即 4GB）。此外，对外部设备的访问或在嵌入式处理器中对片内集成的各种接口设备控制器的访问一般也是同样的方式。具体的映射有如下两种类型。

- ◆ 处理器设计时就已经确定了地址，比如片内内存资源及片内集成的各设备控制器的寄存器。
- ◆ 处理器预定义地址空间，提供相关的片选信号，通过系统总线与外部内存芯片连接。

二、系统的启动方式

这里的启动方式指的是处理器上电后最开始的执行过程，它关系到最初始的程序（一般就是 bootloader）是如何被执行的，这个程序是如何存放到固态存储中的，即烧写程序的方法。对于各种型号的处理器的系统启动方式的差异很大，即使同一款处理器，也往往有多种启动方式，可通过硬件配置进行选择。

三、系统时钟

时钟可以说是处理器的心脏。处理器实质上是一个复杂的时序状态机，其正常运作有赖于时钟提供驱动。一般来说，处理器上电后，并非马上工作在其正常工作时的时钟频率，而是首先工作在一个较低的频率，然后通过指令（初始执行程序）进行配置，切换到正常工作的频率。同样，处理器内的各个接口的正常工作也有赖于时钟系统。因此处理器的时钟管理是系统非常重要的功能，它与处理器的功耗与电源管理等密切相关。

本节的主要内容是基于 HY2410A 开发板，对嵌入式系统中的硬件进行说明。

4.1.1 嵌入式系统架构

如图 4.1 所示是现代嵌入式系统的主要结构，是一个由 S3C2410 作为主控制器的经过简化的嵌入式最小系统。简化的方式是抽出组成系统必需的结构与接口，去掉其他针对不同应用的接口，以便把握最核心的内容。

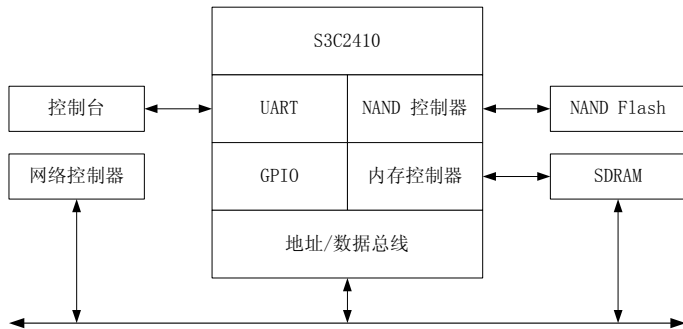


图 4.1 嵌入式最小系统

HY2410A 开发板采用 S3C2410A 芯片作为主控制器，并使用 CS8900A 芯片作为外部扩展的以太网控制器。S3C2410A 芯片本身集成了 ARM 处理器内核、NAND Flash 控制器、内存控制器、GPIO 控制器等部分。

4.1.2 S3C2410A 地址映射

根据硬件的设置，S3C2410A 有两种启动模式，在不同模式下，处理器地址空间的映射是有差别的。HY2410A 开发板采用的是 NAND Flash 启动方式，在这种方式下 S3C2410A 的内存映射方式如图 4.2 所示。地址空间按照 128MB 大小的区块划分，可通过相应的片选信号进行选择。实际上，不同的片选决定了相应的地址空间映射到何种设备上。具体来说，主要的地址映射关系如下。

- ◆ 外接 SDRAM 的地址范围是从 0x30000000 到 0x40000000，可支持的最大内存容量为 256MB。
- ◆ SFR 区域安排为片内控制器的寄存器地址，设计时已经完全确定，不能更改。
- ◆ 从 0x08000000 到 0x30000000 的地址范围划分为 5 个 128MB 的区块，可扩展外接 SROM 类型的存储器。如果硬件上使用 NOR Flash 芯片，可以将这个区块的地址映射到相应的 NOR Flash 存储空间上。

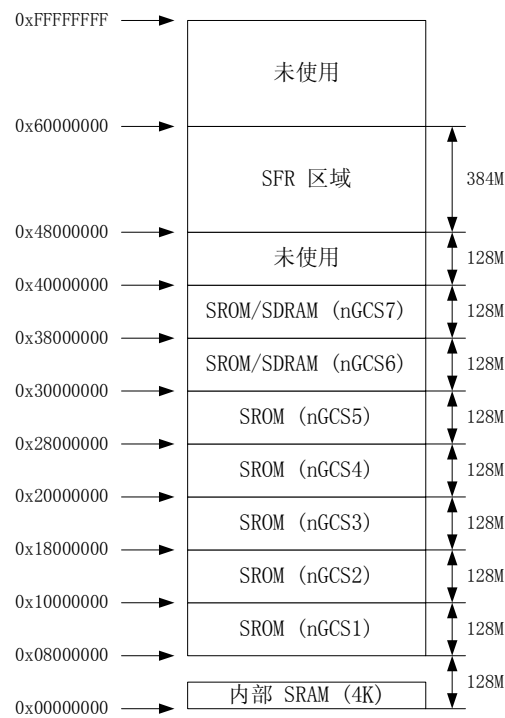


图 4.2 S3C2410A 从 NAND 启动时的内存映射

特别要强调的是，在片内有一块从地址 0x00000000 开始的 4KB 大小的存储空间，它是 SRAM 存储器。由于处理器不能直接执行 NAND Flash 中保存的指令，因此 S3C2410A 处理器在上电复位时，要先从其外接的 NAND Flash 芯片中复制最开始的 4KB 内容到 SRAM 中，然后将 PC 寄存器复位为 0，从地址 0x00000000 处开始执行指令，也就是从这个 4KB 大小的 SRAM 缓

寄存器开始执行指令。

4.1.3 HY2410A 开发板硬件配置

以下将介绍 HY2410A 开发板的硬件系统。

一、SDRAM

HY2410A 开发板使用两片 16 位位宽的 32MB SDRAM 存储器组合形成 32 位容量为 64MB 的内存。

二、存储设备

HY2410A 开发板使用 K9F1208 NAND Flash 芯片，容量为 64MB。

三、以太网控制器

因为 S3C2410A 并不支持以太网接口，所以必须进行外部扩展。HY2410A 开发板使用从 0x18000000 到 0x20000000 的 SROM 地址区域进行扩展，选用以太网控制芯片 CS8900A，将 S3C2410A 的 nGCS3 片选信号外接到以太网控制芯片的片选信号。

4.2 ARM 架构概述

本节将主要介绍 ARM 架构处理器硬件上的一些特点与功能。

4.2.1 ARM 处理器模式与寄存器组

ARM 处理器有 37 个 32 位寄存器，其中包括 30 个通用寄存器、6 个状态寄存器和一个程序计数寄存器，如图 4.3 所示。ARM 处理器中将这 37 个寄存器分成不同的组，在每种工作模式下只能使用其中一组寄存器。

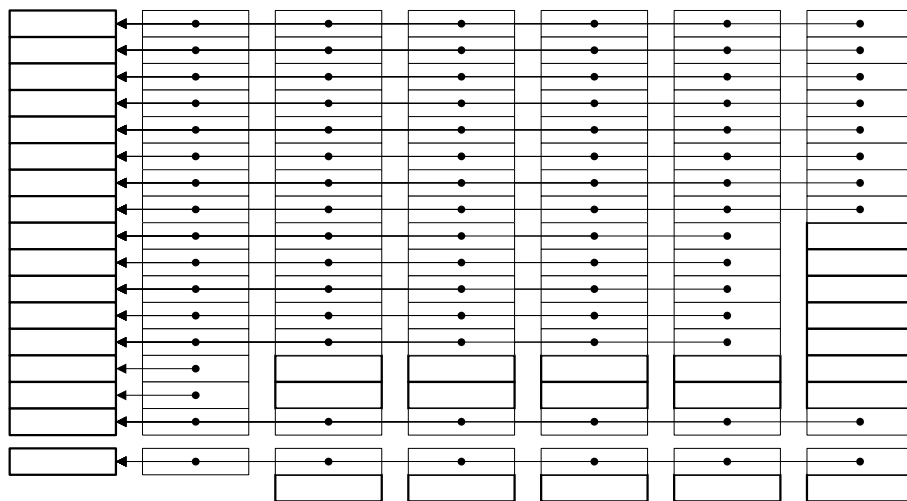


图 4.3 ARM 处理器模式与寄存器组

ARM 处理器共有 7 种模式，其中用户（USR）模式和系统（SYS）模式拥有物理空间上完全相同的寄存器，而其他 5 种异常模式都有一些自己独立的寄存器。

从图中可以看出，在用户模式和系统模式下可以使用 R0 到 R15 和 CPSR（Current Program Status Register，当前程序状态寄存器）共 17 个寄存器；在 FIQ 模式下可以使用 R0 到 R15，CPSR，SPSR（Saved Program Status Register，备份程序状态寄存器）共 18 个寄存器，其中 R8 到 R14 以及 SPSR 寄存器是 FIQ 模式专有的寄存器，其他寄存器和用户模式共用相同的物理寄存器；在 IRQ, SVC, UND, ABT 模式下可以使用的寄存器都是 18 个，包括 R0 到 R15 及 CPSR 和 SPSR，其中 R13, R14, SPSR 是各个模式专有的，其他寄存器和用户模式共用相同的物理寄存器。

ARM 处理器的大部分寄存器是通用寄存器，只有一部分寄存器有特殊用途或人为地约定了特殊的用途。

- ◆ R13 通常用做栈指针寄存器（SP），每一种模式有自己的 R13，所以允许每一种异常都有自己的栈指针。
- ◆ R14 用做链接寄存器（LR），每一种模式有自己的 R14。
- ◆ R15 用做程序计数器（PC），用来保存读取指令的地址。
- ◆ CPSR 存储 ARM 处理器当前的状态和模式标志。
- ◆ SPSR 是异常模式下的 CPSR 的备份寄存器，当一个异常发生时保存当前的 CPSR 值。结合链接寄存器可使处理器返回先前的状态。

这样组织寄存器的好处在于当各种异常发生的时候可以保留一些重要的寄存器数据不被改动，当异常处理程序执行完成之后返回时不会破坏原有的寄存器或状态。

ARM 处理器的通用寄存器可分为未分组寄存器（R0 - R7）、分组寄存器（R8 - R14）和程序计数器 PC（R15），下面将分别进行说明。

一、未分组寄存器（R0 - R7）

在 ARM 处理器中寄存器 R0 - R7 是未分组的，在物理上只有一组，在任何模式下使用 R0 - R7 寄存器指的都是同一个物理寄存器。未分组寄存器没有被系统用于特殊的用途，任何可使用通用寄存器的场合都可以使用未分组寄存器。在异常中断所引起的处理器模式切换时，要对未分组寄存器加以保护（入栈）以防止破坏寄存器中的数据。R0 - R7 在 Thumb 状态下也称为低组寄存器，而 R8 - R15 在 Thumb 状态下称为高组寄存器。

二、分组寄存器（R8 - R14）

R8 - R14 是分组寄存器，具体访问哪个物理寄存器取决于当前的处理器模式。R8 - R12 这组寄存器物理上有两组，在 FIQ 模式下使用自己专有的 R8 - R12，而其他模式使用同一组寄存器。这样的结构设计有利于加快 FIQ 的处理速度。当处理器处于 ARM 指令状态时，R8 - R12 没有任何指定的其他用途，所以当 FIQ 中断到达时，可以不保存这些通用寄存器，也就是说 FIQ 处理程序在保存和恢复现场时可以少处理几个寄存器（R8 - R12），从而提高中断处理的速度。因此 FIQ 模式常被用来处理一些时间紧迫的任务。

分组寄存器 R13 和 R14，分别对应 6 个不同的物理寄存器。其中用户模式和系统模式共用一个，5 种异常模式中分别有自己的 R13 和 R14，比如在 IRQ 模式下的 R13 和 R14 与在用户模式或其他 4 种异常模式下的 R13 和 R14 是不同的物理寄存器。

R13 寄存器在 ARM 中常用做栈指针，称为 SP。因为 ARM 处理器没有专门的入栈和出栈指

令，所以这只是一种习惯用法。也就是说，并没有任何 ARM 指令强制使用 R13 作为栈指针，用户可以使用其他寄存器作为栈指针。每一种异常模式都有自己的 R13，在使用时要分别对其进行初始化，以保证在相应模式下能正确地进行入栈和出栈操作。另外需要注意，在 Thumb 指令集中，有一些指令强制使用 R13 作为栈指针，如栈操作指令。R13 也可以作为通用寄存器使用。

R14 寄存器又被称为链接寄存器，在 ARM 体系结构中 R14 的特殊用途有两种：一是用来保存子程序返回地址，二是当异常发生时 R14 中保存的值等于异常发生时 PC 的值减 4（或者减 2），因此在各种异常模式下可以根据 R14 的值返回到异常发生前的相应位置继续执行。

当通过 bl 或 blx 指令调用子程序时，硬件自动将子程序返回地址保存在 R14 寄存器中。在子程序返回时，把 R14 的值复制到程序计数器 PC 即可实现子程序返回。例如可以使用如下指令从子程序中返回：

```
mov pc, lr
```

或者：

```
bx lr
```

另外，也可以在子程序入口处使用如下的指令将 LR 保存到栈中：

```
stmfd sp!, {r0-r7, lr}
```

这条指令将 R0 - R7 这 8 个寄存器以及 LR 放入栈中。

与此相对应，在子程序返回时，使用如下指令做出栈操作，实现从子程序返回：

```
ldmfd sp!, {r0-r7, lr}
```

R14 还用于从异常返回。当异常发生时，寄存器 R14 被设置成返回地址（等于异常发生时 PC 的值减 4 或者减 2）。在不同异常模式下，R14 保存的值并不是异常返回的真正地址，而是有一个常数的偏移量。例如用下面的指令可以从 FIQ 异常中返回：

```
subs pc, lr, #4
```

R14 也可以作为通用寄存器使用。

三、程序计数器（R15）

ARM 处理器的寄存器 R15 被用做程序计算器 PC。R15 可以作为通用寄存器使用，但很多特殊的指令在使用 R15 时有限制。当违反了这些指令的使用限制时，指令的执行结果是不可预知的。R15 保存处理器取指令的地址，改变 R15 的值会引起程序执行流程的改变。

由于 ARM 指令都是 32 位并且按字对齐的，所以当处理器处于 ARM 指令状态时 R15 的低两位永远是 0；而 Thumb 指令是 16 位并且按 2 字节对齐的，所以当处理器处于 Thumb 状态下时 R15 的最低位永远是 0。

由于 ARM 的流水线机制，指令读取的 R15 的值是当前正在执行的指令地址加上 8 个字节。读 PC 主要用于快速地对临近的指令或数据进行位置无关寻址，包括程序中的位置无关分支。

需要注意的是，当使用指令 str 或 stm 对 R15 进行保存时，保存的可能是当前指令地址加 8 或当前指令地址加 12。到底是哪种方式，取决于芯片的具体设计方式。当然，在同一个芯片中，只能采用一种方式，要么保存当前指令地址加 8，要么保存当前指令地址加 12。程序开发人员应



尽量避免使用 `str` 或 `stm` 指令来对 `R15` 进行操作。当不可避免要使用这种方式时，可以先通过一小段程序来确定所使用的芯片是使用哪种方式实现的，例如：

```
sub r1, pc, #4 @ r1 中存放 str 指令地址
str pc, [r0] @ 用 str 指令将 pc 保存到 r0 指向的地址单元中
ldr r0, [r0] @ 读取 str 指令地址 + 偏移量的值
sub r0, r0, r1 @ str 指令地址 + 偏移量的值减去 str 指令的地址得到偏移量
```

当用指令修改 `R15` 的值时，如果修改成功，它将使程序跳转到该地址执行。写入 `R15` 的地址值应满足低两位为 0 的要求。如果不满足这个要求，具体的处理根据 `ARM` 版本的不同也有所不同。

- ◆ 对于 `ARM` 版本 3 以及更低的版本，写入 `R15` 的地址值低两位被忽略。
- ◆ 对于 `ARM` 版本 4 以及更高的版本，将会产生不可预知的结果。
- ◆ 对于 `Thumb` 指令集来说，处理器将忽略 `R15` 的最低位。

有些指令对 `R15` 的操作有特殊的要求。比如，指令 `BX` 利用寄存器的最低位来确定需要跳转到的子程序是 `ARM` 指令还是 `Thumb` 指令。

4.2.2 ARM 异常与异常向量表

`ARM` 处理器支持 7 种异常，如表 4.1 所示。发生不同的异常时处理器将切换到某个异常模式，并跳转到相应的异常入口地址处执行。异常具有不同的优先级，有固定的异常入口地址。当一个异常发生时，相应的 `R14 (LR)` 存储的是异常返回地址，`SPSR` 存储的是异常发生前状态寄存器 `CPSR` 的值。

`ARM` 处理器对异常的入口地址提供了两种可选的配置，即低端地址和高端地址，分别从地址 `0x00000000` 和 `0xFFFF0000` 处开始。一般来说，可以通过系统控制器在这两种配置中进行选择。`Linux` 内核使用的是高端入口地址。

表 4.1 ARM 异常类型

异常类型	处理器模式	低端入口地址	高端入口地址	优先级	异常返回指令
复位	SVC	0x00000000	0xFFFF0000	1 (最高)	无
未定义指令	UND	0x00000004	0xFFFF0004	6 (最低)	<code>movs pc, lr</code>
软件中断	SVC	0x00000008	0xFFFF0008	6	<code>movs pc, lr</code>
指令预取中止	ABT	0x0000000C	0xFFFF000C	5	<code>subs pc, lr, #4</code>
数据中止	ABT	0x00000010	0xFFFF0010	2	<code>subs pc, lr, #8</code>
中断	IRQ	0x00000018	0xFFFF0018	4	<code>subs pc, lr, #4</code>
快速中断	FIQ	0x0000001C	0xFFFF001C	3	<code>subs pc, lr, #4</code>



地址偏移量 `0x14` 处没有定义。

下面将具体说明 7 种异常的含义及异常发生的条件。



一、复位

处理器复位信号有效时,产生复位异常,处理器进入管理模式并禁止中断。程序跳转到复位异常处理程序处执行。

二、未定义指令

ARM 处理器或协处理器遇到不能处理的指令时,产生未定义指令异常,可使用该异常机制进行软件仿真或断点调试。

三、软件中断

这种异常通过软件上执行指令 `swi` 而触发,可用于用户模式下的程序调用特权操作指令,一般使用此异常机制实现操作系统的功能调用。

四、指令预取中止

若处理器预取指令的地址不存在,或该地址不允许当前指令访问,则由 MMU (Memory Management Unit, 内存管理单元)向处理器发出中止信号,于是在预取的指令被执行时,产生指令预取中止异常。

五、数据中止

若处理器数据访问指令的地址不存在,或该地址不允许当前指令访问时,产生数据中止异常。由 MMU 向处理器发出中止信号,在后续的任何指令或异常发生之前,数据中止异常发生。

六、中断

当处理器外部中断请求信号有效,且 CPSR 寄存器中的 I 位为 0 时,产生中断异常。系统的外设可以通过该异常请求中断服务。中断异常的优先级比快速中断异常低,当进入快速中断异常处理时,会屏蔽掉中断。

七、快速中断

当处理器快速中断请求信号有效时,且 CPSR 寄存器中的 F 位为 0 时,产生快速中断异常。异常发生时,程序正常的执行流程就被打断,此时 ARM 处理器将执行如下动作。

- step 1** 在相应模式的 LR 寄存器中保存返回地址。
- step 2** 复制 CPSR 寄存器的内容到相应模式的 SPSR 寄存器。
- step 3** 设置合适的 CPSR 标志位。
- step 4** 改变处理器状态。
- step 5** 进入到相应异常模式。
- step 6** 禁止相应的中断。
- step 7** 设置 PC 寄存器的值到相关的异常向量地址,跳转到异常处理代码处执行。

4.2.3 程序状态寄存器

CPSR 寄存器可以在任何处理器模式下被访问。而每一种异常模式下都有一个专用的物理状态寄存器 SPSR,当特定的异常发生时,这个寄存器用于存放异常发生前 CPSR 寄存器的内容。在异常退出时,可以用 SPSR 中保存的值来恢复 CPSR。CPSR 寄存器所保存的数据格式如图 4.4 所示。

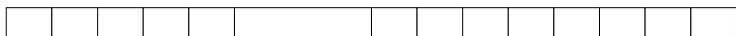


图 4.4 程序状态寄存器格式

这 32 个比特位可分为 4 类：条件标志位、控制位、模式标识位和保留位。下面将参照图 4.4 来解释各个比特位的含义。

一、条件标志位

N (Negative), Z (Zero), C (Carry) 及 V (oVerflow) 统称为条件标志位。大部分的 ARM 指令可以依据 CPSR 中的这些标志位来选择性地执行。各条件标志位的具体含义如表 4.2 所示。

表 4.2 CPSR 条件标志位及其含义

标志位	含义
N	本位设置成当前指令运算结果的第 31 位的值，当两个补码表示的有符号整数运算时，N = 1 表示运算的结果为负数，N = 0 表示结果为正数或零
Z	Z = 1 表示运算结果为零，Z = 0 表示运算结果不为零；对于 CMP 指令，Z = 1 表示进行比较的两个数大小相等
C	在加法指令中（包括比较指令 CMN），结果产生进位了，则 C = 1，表示无符号数运算发生上溢出，其他情况下 C = 0；在减法指令中（包括比较指令 CMP），结果产生借位了，则 C = 0，表示无符号数运算发生下溢出，其他情况下 C = 1；对于包含移位操作的非加减法运算指令，C 中包含最后一次被溢出的位的数值，对于其他非加减法运算指令，C 位的值通常不受影响
V	对于加减法运算指令，当操作数和运算结果为二进制的补码表示的带符号数时，V = 1 表示符号位溢出，其他的指令通常不影响 V 位

二、Q 标志位

在带 DSP 指令扩展的 ARM v5E 及以上版本的处理器中，Q (Sticky Overflow) 标志位主要用于指示增强的 DSP 指令是否发生了溢出。

三、控制位

I, F, T 三个比特位称为控制位，当异常发生时这些位发生变化。在特权级的处理器模式下，软件可以修改这些控制位。各个控制位的含义如下。

- ◆ I 为中断禁止位，当 I = 1 时禁止中断。
- ◆ F 为快速中断禁止位，当 F = 1 时禁止快速中断。
- ◆ T 用于控制指令执行的状态，即当前执行的指令是 ARM 指令还是 Thumb 指令。

通常在中断服务程序中可以通过置位 I 和 F 标志位来禁止中断，但是在中断服务程序返回前应恢复原来 I 和 F 位的值。

对于不同版本的 ARM 处理器，T 控制位的含义也是不同的，说明如下。

- ◆ 对于 ARM v3 及更低的版本和 ARM v4 的非 T 系列版本的处理器，因为不支持 Thumb 指令，所以 T 始终为 0。
- ◆ 对于 ARM v4 及更高版本的 T 系列处理器，当 T = 0 时表示执行 ARM 指令，当 T = 1 时表示执行 Thumb 指令。

- ◆ 对于 ARMv5 及更高的版本的非 T 系列处理器，当 $T = 0$ 时表示执行 ARM 指令，当 $T = 1$ 时表示强制下一条执行的指令产生未定义指令异常。

四、模式标识位

M4 - M0 称为处理器模式标识位，它们的值用于表示处理器处于何种模式，具体说明如表 4.3 所示。

表 4.3 处理器模式标识位说明

M4 - M0 的值（二进制）	处理器模式
10000	USR
10001	FIQ
10010	IRQ
10011	SVC
10111	ABT
11011	UND
11111	SYS

五、保留位

程序状态寄存器的第 8 位到第 26 位为保留位，可能用于将来 ARM 版本的扩展，在程序中不应该操作这些位。

4.2.4 大端与小端存储格式

存储格式指的是处理器解释存储器中数据的方式，或者说存储器中的数据与处理器寄存器中数据（机器字）的对应方式，可分为以下两种。

- ◆ 小端（Little endian）格式：机器字的低位字节存放在存储空间的低地址。
- ◆ 大端（Big endian）格式：机器字的高位字节存放在存储空间的低地址。

ARM 处理器既支持使用小端格式也支持使用大端格式，这可以通过系统控制器进行设置。一般使用小端格式，包括 Linux 系统。

图 4.5 说明了存储格式对处理器访问整型数据的影响。开始时，寄存器 r0 中存放着数据 0x11223344，将这个数据保存到内存中某个地址后，如果是小端格式，则字节 0x44 放在起始地址处，如果是大端格式，则字节 0x11 放在起始地址处。然后从同一地址读一个字节到寄存器，小端格式下读到的是 0x44，而大端格式下读到的是 0x11。

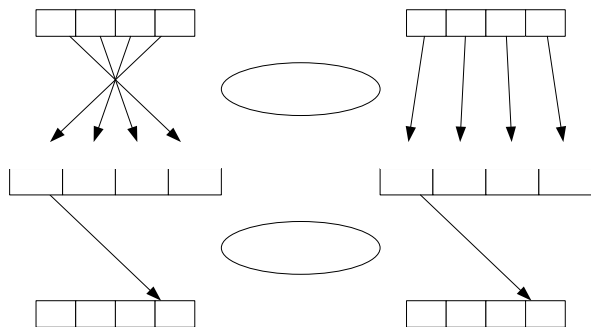


图 4.5 大端与小端存储方式



4.3 ARM 指令集概述

ARM 指令集中的指令可分为以下七大类：

- ◆ 数据处理指令
- ◆ 存储器访问指令
- ◆ 分支指令
- ◆ 软中断指令
- ◆ 程序状态寄存器传送指令
- ◆ 乘法指令
- ◆ 协处理器指令

ARM 指令机器码的最高 4 个比特位称为条件码，用于控制指令在某种条件成立时才真正执行，否则不做任何操作。

大部分数据处理指令能够按照其执行结果更新 CPSR 寄存器中的 4 个条件标志位（N，Z，C，V），但是否更新 CPSR 寄存器的值也是受指令机器码中的一个比特位控制的。

有些 ARM 处理器还支持一种 16 位长度的指令集，称为 Thumb 指令。Thumb 指令一般用在需要提高指令密度的地方。限于篇幅，本章中不介绍 Thumb 指令，并且全书的例程中没有用到 Thumb 指令。

在介绍指令格式时，我们将用尖括号 < > 表示可选的部分，其中多个用竖线隔开的部分表示任选其一，而用圆括号中多个竖线隔开的部分表示必须在这些部分中任选其一。

4.3.1 ARM 汇编指令格式

ARM 汇编指令的基本格式如下：

```
opcode<cond> rd, rn<, operand2>
```

对各个部分的含义解释如下。

- ◆ opcode：指令助记符，表示要进行的操作，如 mov，add 等。
- ◆ cond：条件码助记符，默认条件是 al。
- ◆ rd：目标寄存器。
- ◆ rn：第一操作数所在的寄存器。
- ◆ operand2：第二操作数。

下面将对条件码和第二操作数这两个重要部分进行说明。

一、条件码

条件码是 ARM 指令的一大特色，使用条件码可以实现高效的分支处理，提高代码效率。可以使用的条件码及含义如表 4.4 所示。

表 4.4 指令中的条件码

条件码	助记符	执行条件	含义
0000	eq	Z = 1	相等
0001	ne	Z = 0	不相等



(续表)

条件码	助记符	执行条件	含义
0010	cs/hs	$C = 1$	无符号数大于或等于
0011	cc/lo	$C = 0$	无符号数小于
0100	mi	$N = 1$	负数
0101	pl	$N = 0$	正数或零
0110	vs	$V = 1$	溢出
0111	vc	$V = 0$	没有溢出
1000	hi	$C = 1, Z = 0$	无符号数大于
1001	ls	$C = 0, Z = 1$	无符号数小于或等于
1010	ge	$N = V$	有符号数大于或等于
1011	lt	$N = V$	有符号数小于
1100	gt	$Z = 0, N = V$	有符号数大于
1101	le	$Z = 1, N \neq V$	有符号数小于或等于
1110	al	任意	无条件执行 (指令默认条件)

二、第二操作数

ARM 指令的第二操作数用法比较灵活,有如下几种情况。

- ◆ 立即数,其有效位数只能有 8 位,但可以进行一定范围内的移位操作,因此其取值范围并不局限在 0 ~ 255 之内。
- ◆ 寄存器,这时操作数即为寄存器保存的数值。
- ◆ 寄存器移位,这时操作数是寄存器值移位后的结果 (寄存器本身的值不变)。

当第二操作数是立即数时,ARM 机器指令使用 8 个比特表示它的有效位数,而另外 4 个比特表示循环右移位数的一半,也就是说,右移的位数取值是 0 ~ 30 之间的偶数。实际操作数的值是这 8 个有效位数进行循环移位的结果,这样就扩大了立即数的取值范围。

但这并不表示任意的 32 位立即数都可以作为第二操作数,因此在书写指令时需要注意第二操作数的取值,如果它无法用 8 位二进制数循环移位偶数次来表达,则这条指令不可能实现。

当第二操作数使用寄存器移位时,可能采取的移位方式如表 4.5 所示。

表 4.5 第二操作数移位方式

助记符	功能	说明
lsl	逻辑左移	低端空出的位补 0
lsr	逻辑右移	高端空出的位补 0
asr	算术右移	保持符号位不变,即如果原值为正数,则高端空出的位补 0,否则补 1
ror	循环右移	低端移出的位填入高端空出的位
rrx	扩展循环右移	右移一位,高端空出的位用原 C 标志值填充,低端移出的位是新的 C 标志值

这些移位操作的实质可由图 4.6 形象地说明。

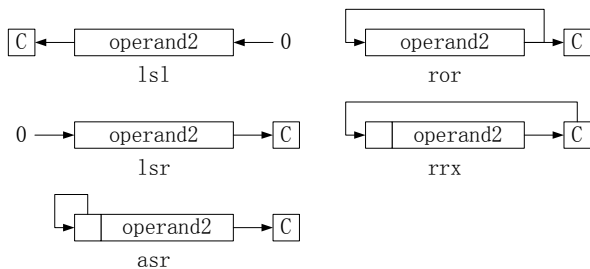


图 4.6 移位操作

移位数既可以是一个常数，也可以是一个寄存器，实际移位的数目是寄存器值的低 8 位。

4.3.2 数据处理指令

数据处理指令大致可分为以下几类：数据传送指令、算术运算指令、逻辑运算指令和比较指令。数据处理指令只能对寄存器的内容进行操作。

4.3.2.1 数据传送指令

mov 指令和 **mvn** 指令用于向寄存器传送数据，它们的格式如下：

```
mov<cond><s> rd, operand2
mvn<cond><s> rd, operand2
```

其各个部分的含义解释如下。

- ◆ **cond**: 条件码助记符。
- ◆ **s**: 表示传送的数据将影响 CPSR 寄存器中的条件标志位，默认不影响。
- ◆ **rd**: 目标寄存器。
- ◆ **operand2**: 第二操作数，放入目标寄存器的数据，其格式遵循第二操作数的一般原则。

mov 指令将第二操作数的值放入目标寄存器，**mvn** 指令则将第二操作数的值取反后放入目标寄存器。

因为 ARM 没有专门的移位指令，实际上可以用 **mov** 指令结合寄存器移位方式的第二操作数来实现。

当 **movs** 或 **mvns** 的目标寄存器是 R15 时，这条指令将同时把 SPSR 寄存器的值复制到 CPSR。其他的数据处理指令也有这样的特点。这种设计是为了从异常模式中返回，在用户模式或系统模式下不应该使用这样的指令。

4.3.2.2 算术和逻辑运算指令

算术和逻辑运算的指令都具有如下的格式：

```
opcode<cond><s> rd, rn, operand2
```

其各个部分的含义解释如下。

- ◆ opcode: 指令助记符。
- ◆ cond: 条件码助记符。
- ◆ s: 表示计算的结果将影响 CPSR 寄存器中的条件标志位, 默认不影响。
- ◆ rd: 目标寄存器, 计算结果保存在此寄存器中。
- ◆ rn: 第一操作数寄存器, 它的值是第一个操作数。
- ◆ operand2: 第二操作数, 其格式遵循第二操作数的一般原则。

可以使用的指令助记符及相应的功能如表 4.6 所示。

表 4.6 算术和逻辑运算指令助记符及相应的功能

指令助记符	功能
add	加法指令, $rn + operand2 \rightarrow rd$
sub	减法指令, $rn - operand2 \rightarrow rd$
rsb	逆向减法指令, $operand2 - rn \rightarrow rd$
adc	带进位加法指令, $rn + operand2 + C \rightarrow rd$
sbc	带借位减法指令, $rn - operand2 + C - 1 \rightarrow rd$
rsc	带借位逆向减法指令, $operand2 - rn + C - 1 \rightarrow rd$
and	与指令, $rn \& operand2 \rightarrow rd$
orr	或指令, $rn operand2 \rightarrow rd$
eor	异或指令, $rn \wedge operand2 \rightarrow rd$
bic	位清除指令, $rn \& \sim operand2 \rightarrow rd$

在带进位/借位的加/减法运算中, 参与计算的不仅是两个操作数, 还要考虑 C 标志位的值。这里要注意, 进行加法运算时, C 标志位为 1 说明发生进位; 进行减法运算时, C 标志位为 0 说明发生借位。

4.3.2.3 比较指令

比较指令的特点是运算的结果不保存, 只是影响 CPSR 寄存器中的条件标志位, 以便后续指令根据条件标志位进行执行。比较指令的格式如下:

```
opcode<cond> rn, operand2
```

其各个部分的含义解释如下。

- ◆ cond: 条件码助记符。
- ◆ rn: 第一操作数寄存器, 它的值是第一个操作数。
- ◆ operand2: 第二操作数, 其格式遵循第二操作数的一般原则。

比较指令因为不保存结果, 所以不需要写出目标寄存器。可以使用的指令助记符及相应的功能如表 4.7 所示。

表 4.7 比较指令助记符及相应的功能

指令助记符	功能	类比的算术或逻辑指令
cmp	rn 的值减去 operand2，根据结果修改标志位	subs
cmn	rn 的值与 operand2 相加，根据结果修改标志位	adds
tst	rn 的值与 operand2 按位与，根据结果修改标志位	ands
teq	rn 的值与 operand2 按位异或，根据结果修改标志位	eors

比较指令实际上也是在做算术或逻辑运算，只是运算的结果不保存到寄存器而已。比较指令必须与后续指令的条件码搭配使用才有意义。

4.3.3 存储器访问指令

ARM 处理器具有 RISC 处理器的加载/存储体系结构，对存储器的访问只能使用加载和存储指令实现。ARM 处理器的加载和存储指令可以对字、半字、无符号字节及有符号字节这几种类型的数据进行操作。ARM 处理器还拥有批量加载和存储的指令，可以用一条指令加载或存储多个寄存器的内容，大大提高了指令的效率。

4.3.3.1 ldr 和 str 指令

ldr 指令用于读取存储器中的数据到寄存器，str 指令用于将寄存器中的数据保存到存储器中。指令的格式如下：

```
ldr<cond><s><h|b><t> rd, [rn<, offset>]<!>
str<cond><h|b><t> rd, [rn<, offset>]<!>
ldr<cond><s><h|b><t> rd, [rn]<, offset>
str<cond><h|b><t> rd, [rn]<, offset>
```

其各个部分的含义解释如下。

- ◆ cond: 条件码助记符。
- ◆ s: 只用于 ldr 指令且必须与 h 或 b 一起使用，表示加载时高位字节以有符号方式扩展，即原数据是正数或 0 时扩展为 0，原数据是负数时扩展为 1，否则都扩展为 0。
- ◆ h: 表示对半个机器字（16 位）操作。用于 ldr 指令时，取存储器中的两个字节并扩展为一个机器字后放入寄存器；用于 str 指令时，寄存器的高 16 位无效。
- ◆ b: 表示对字节操作。用于 ldr 指令时，取存储器中的一个字节并扩展为一个机器字后放入寄存器；用于 str 指令时，寄存器的高 24 位无效。
- ◆ t: 表示将操作视为在用户模式下，在地址为前索引偏移方式时不可用。
- ◆ rd: 目标寄存器。
- ◆ rn: 基址寄存器。
- ◆ offset: 地址偏移量。

ldr 和 str 的寻址模式非常灵活。存储器地址由两部分组成：一部分为基址寄存器 rn，可以是任意一个通用寄存器，另一部分为地址偏移量 offset，而地址偏移量又有以下几种格式。

- ◆ 立即数：可以是正数或负数。

- ◆ 寄存器：寄存器的值作为偏移量，可以加在基址上，也可以从基址减去（前面加负号）。
- ◆ 寄存器移位：此时移位数是一个常数，寄存器移位后的值可以加在基址上，也可以从基址减去（寄存器前加负号）。

从寻址时的地址计算方法分，`ldr` 和 `str` 指令有以下几种形式。

- ◆ 零偏移：地址偏移量为 0。
- ◆ 前索引偏移：在数据传送之前将偏移量与基址相加，其结果作为传送数据的存储器，若在方括号后加上符号 `!`，则同时将相加的结果写回基址寄存器。这时偏移量写在方括号中。
- ◆ 后索引偏移：先以基址作为地址进行数据传送，然后将偏移量与基址相加写回基址寄存器。这时偏移量写在方括号后。
- ◆ PC 相对偏移：以 PC 作为基址寄存器。

当使用前索引偏移和后索引偏移时，基址寄存器不能是 R15，即 PC。对于 PC 相对偏移的情况，指令的写法不遵循上述形式，如：

```
ldr r0, label
```

其中 `label` 是程序中的一个标号，编译器将自动计算标号处与当前 PC 值的偏移量并生成一条前索引的存储指令。



访问存储器时必须注意地址的对齐，即读写 32 位数据时，地址必须是 4 的整数倍；读写 16 位数据时，地址必须是 2 的整数倍。

4.3.3.2 ldm 和 stm 指令

`ldm` 指令用于读取存储器中连续的多个数据到多个寄存器，`stm` 指令用于将多个寄存器中的数据保存到存储器的连续空间中。指令的格式如下：

```
ldm<cond>(i|d)(a|b) rn<!>, {reglist}<^>
stm<cond>(i|d)(a|b) rn<!>, {reglist}<^>
```

其各个部分的含义解释如下。

- ◆ `cond`：条件码助记符。
- ◆ `i`：每次传送基址加 4。
- ◆ `d`：每次传送基址减 4。
- ◆ `a`：传送一个寄存器之后基址变化（加 4 或减 4）。
- ◆ `b`：传送一个寄存器之前基址变化（加 4 或减 4）。
- ◆ `rn`：基址寄存器。
- ◆ `reglist`：要传送的寄存器列表。

基址寄存器后面如果有符号 `!` 则表示变化后的基址写回基址寄存器。基址寄存器不可以是 PC。

寄存器列表中的多个寄存器用逗号隔开，如果是多个连续的寄存器，可使用连字符，如：

```
stmia r0!, {r3 - r5, lr} ; 保存 r3, r4, r5 和 lr 到 r0 指向的存储空间中
```

列表中的寄存器不能重复，并且最好按编号从小到大排列。

符号 ^ 的作用比较特殊，它不能在用户模式和系统模式下使用，如果在 ldm 指令的寄存器列表中有 PC 并且使用了 ^ 符号，则除了传送所指定的寄存器外，还要将 spsr 复制到 cpsr 中，这可以用来从异常模式返回。如果使用了 ^ 符号并且寄存器列表中不包含 PC，操作的是用户模式下的寄存器而不是当前模式下的。

当基址寄存器是 sp 时，这些指令的作用就是入栈和出栈操作。鉴于栈操作的重要性，对指令的格式重新定义如下：

```
ldm<cond>(f|e)(d|a) sp<!>, {reglist}<^>  
stm<cond>(f|e)(d|a) sp<!>, {reglist}<^>
```

其中 f, e, d 和 a 四个字母的含义如下。

- ◆ f: 满栈，即栈指针总是指向最后入栈的元素。
- ◆ e: 空栈，即栈指针总是指向将要入栈的元素。
- ◆ d: 递减堆栈，即入栈操作使栈指针递减。
- ◆ a: 递增堆栈，即入栈操作使栈指针递增。



这些指令只对 32 位长度的数据进行操作，地址的低两位将被忽略。

4.3.3.3 swp 指令

swp 指令用于交换寄存器和存储器中的值，格式如下：

```
swp<cond><b> rd, rm, [rn]
```

其各个部分的含义解释如下。

- ◆ cond: 条件码助记符。
- ◆ b: 表示对字节数据进行操作。
- ◆ rd: 目标寄存器，从存储器中读到的数据放入此寄存器。
- ◆ rm: 第一操作数寄存器，此寄存器中的值写入存储器。
- ◆ rn: 基址寄存器，存储器的地址。

当 rd 和 rm 相同时，实质上就成了交换操作。

4.3.4 分支指令

在 ARM 汇编中可以使用两种方式实现程序的跳转：一种是直接修改 PC 寄存器的值实现跳转，另一种则是使用分支指令，如 b, bl, bx 等。

4.3.4.1 b 指令

b 指令使程序跳转到指定的地址，其格式如下：

```
b<cond> label
```

其中 **cond** 是条件码助记符，**label** 则是程序中的一个标号。b 指令的实质就是给 PC 寄存器的值加上或减去一个偏移量，因此是一种相对跳转。ARM 机器指令中为跳转指令留出了 24 个比特位作为偏移量，由于 ARM 指令的低两位一定是 0，不必放在指令中，实际跳转的偏移量是这个 24 位数左移两位，也就是乘以 4，因此 b 指令跳转的目标必须在当前指令的前后 32MB 范围内。

4.3.4.2 bl 指令

bl 指令首先将下一条指令的地址复制到 lr 寄存器中，然后使程序跳转到指定的地址，因此常用来调用子程序。bl 指令的格式如下：

```
bl<cond> label
```

其中 **cond** 是条件码助记符，**label** 则是程序中的一个标号。与 b 指令类似，bl 指令也是一种相对跳转，并且跳转的目标也被限制在当前指令的前后 32MB 范围内。

4.3.4.3 bx 指令

bx 指令是带状态切换的跳转指令，其格式如下：

```
bx<cond> rm
```

其中 **cond** 是条件码助记符，**rm** 则是一个寄存器。bx 指令将把 rm 的值复制到 PC 寄存器实现跳转，并且如果 rm 的最低位是 1，则同时将 CPSR 寄存器中的 T 控制位置为 1，也就是处理器将进入执行 Thumb 指令状态；如果 rm 的最低位是 0，则同时将 CPSR 寄存器中的 T 控制位清零，也就是处理器将进入执行 ARM 指令状态。

4.3.5 软中断指令

swi 是软中断指令，用于产生软中断异常，其格式如下：

```
swi<cond> no
```

其中 **cond** 是条件码助记符，**no** 是一个 24 位的立即数。

swi 指令执行后，下一条指令的地址被保存到 LR 寄存器，CPSR 寄存器的值保存到管理模式（SVC）的 SPSR 寄存器中，处理器切换到管理模式，执行流程转移到软中断异常入口。

swi 在操作系统中用来实现系统调用。no 这个立即数是被处理器忽略的，但软件可以利用它来传递系统调用的功能号。Linux 上的传统 ABI（Application Binary Interface，应用程序二进制接口）就采用这种方式。另外最新的 EABI（Embedded Application Binary Interface，嵌入式应用程序二进制接口）使用 R7 寄存器来传递功能号，而 swi 指令中的立即数被设为 0。

4.3.6 程序状态寄存器传送指令

mrs 指令用于读程序状态寄存器，它是唯一可以读寄存器 **CPSR** 或 **SPSR** 的指令，其格式如下：

```
mrs<cond> rd, psr
```

其各个部分的含义解释如下。

- ◆ **cond**: 条件码助记符。
- ◆ **rd**: 目标寄存器，程序状态寄存器的值将放入此寄存器。
- ◆ **psr**: **cpsr** 或 **spsr**。

这条指令中目标寄存器不能是 **PC**。

msr 指令用于写程序状态寄存器，它是唯一可以直接写寄存器 **CPSR** 或 **SPSR** 的指令，其格式如下：

```
msr<cond> psr_fields, (im8|Rm)
```

其各个部分的含义解释如下。

- ◆ **cond**: 条件码助记符。
- ◆ **psr**: **cpsr** 或 **spsr**。
- ◆ **fields**: 指定写入的区域，字母 **c** 表示写入第 0 - 7 位（控制位），字母 **x** 表示写入第 8 - 15 位，字母 **s** 表示写入第 16 - 23 位，字母 **f** 表示写入第 24 - 31 位（标志位）。可以同时有多个字母。
- ◆ **im8**: 8 位立即数。
- ◆ **Rm**: 寄存器。



只有在特权模式（非用户模式）下才能修改程序状态寄存器。

msr 指令不能直接修改 **CPSR** 寄存器的 **T** 控制位来切换 **ARM/Thumb** 指令状态，必须用 **bx** 指令。

mrs 与 **msr** 指令搭配使用，即可实现修改 **CPSR** 或 **SPSR** 寄存器中一个或多个比特位的功能，从而可以用来进行处理器模式切换以及禁用/使能中断等操作。

4.3.7 乘法指令

ARM 中的乘法指令按结果的位数可分为 32 位乘法指令和 64 位乘法指令。

4.3.7.1 32 位乘法指令

mul 指令用于两个寄存器值相乘，其格式如下：

```
mul<cond><s> rd, rm, rs
```

它的功能是将 `rm` 和 `rs` 寄存器的值相乘并将结果的低 32 位保存到 `rd` 寄存器中。其中 `cond` 是条件码助记符，而 `s` 的含义与算术指令中相同，表示结果是否影响条件标志位。

`mmla` 指令是乘法和加法的组合，其格式如下：

```
mmla<cond><s> rd, rm, rs, rn
```

它的功能是将 `rm` 和 `rs` 寄存器的值相乘并加上 `rn` 寄存器的值，结果的低 32 位保存到 `rd` 寄存器中。

这些指令的目标寄存器不能是 `PC`。

4.3.7.2 64 位乘法指令

64 位乘法指令的格式如下：

```
umull<cond><s> rdlo, rdhi, rm, rs
smull<cond><s> rdlo, rdhi, rm, rs
```

其各个部分的含义解释如下。

- ◆ `cond`：条件码助记符。
- ◆ `s`：表示结果是否影响条件标志位。
- ◆ `rdlo`：用于存放结果的低 32 位的目标寄存器。
- ◆ `rdhi`：用于存放结果的高 32 位的目标寄存器。
- ◆ `rm, rs`：寄存器，两者的值用于相乘。

这两条指令的功能都是将 `rm` 和 `rs` 寄存器的值相乘，并将结果的低 32 位放入 `rdlo` 寄存器，高 32 位放入 `rdhi` 寄存器。不同点在于，`umull` 指令将所有操作数和结果作为无符号数处理，而 `smull` 指令将所有操作数和结果作为有符号数处理。

64 位乘法与加法组合指令的格式如下：

```
ummlal<cond><s> rdlo, rdhi, rm, rs
smmlal<cond><s> rdlo, rdhi, rm, rs
```

其各个部分的含义与 `umull` 和 `smull` 指令相同，从功能上来说，它们会将寄存器 `rm` 和 `rs` 的值相乘的结果加上由 `rdhi` 和 `rdlo` 组成的 64 位整数作为最终结果并保存在 `rdhi` 和 `rdlo` 中。

使用这些指令时，目标寄存器不能是 `PC`，并且 `rdlo`、`rdhi` 和 `rm` 三个寄存器必须各不相同。

4.3.8 协处理器指令

ARM 架构支持协处理器，相应地有对协处理器进行控制的命令。在这些指令中一般都需要指明所操作的协处理器。ARM 可支持 16 个协处理器，编号从 0 到 15，指令中用字母 `p` 加上编号来表示一个协处理器。

4.3.8.1 cdp 指令

cdp 指令用于通知协处理器执行指定的操作，其格式如下：

```
cdp<cond> pn, opcode1, crd, crn, crm<, opcode2>
```

其各个部分的含义解释如下。

- ◆ **cond**: 条件码助记符。
- ◆ **pn**: 协处理器。
- ◆ **opcode1**: 协处理器的第一操作码，它的含义由协处理器解释。
- ◆ **crd**: 协处理器的目标寄存器。
- ◆ **crn**: 协处理器的第一操作数寄存器。
- ◆ **crm**: 协处理器的第二操作数寄存器。
- ◆ **opcode2**: 协处理器的第二操作码，它的含义由协处理器解释。

如果协处理器不能成功执行指定的操作，就会产生未定义指令异常。

4.3.8.2 ldc 和 stc 指令

ldc 指令用于从存储器读取数据到协处理器的寄存器，而 **stc** 指令则是将协处理器寄存器中的数据保存到存储器中。它们的格式如下：

```
ldc<cond><l> pn, crd, addr  
stc<cond><l> pn, crd, addr
```

其各个部分的含义解释如下。

- ◆ **cond**: 条件码助记符。
- ◆ **l**: 表示传送的是长整数。
- ◆ **pn**: 协处理器。
- ◆ **cd**: 协处理器的目标寄存器。
- ◆ **addr**: 地址，地址的格式与前述 **ldr** 和 **str** 指令的地址相同。

数据的传送由协处理器来控制。如果协处理器不能成功执行指定的操作，就会产生未定义指令异常。

4.3.8.3 mcr 和 mrc 指令

mcr 指令用于从 **ARM** 处理器的寄存器向协处理器传送数据，**mrc** 指令则用于从协处理器寄存器向 **ARM** 处理器的寄存器传送数据。它们的格式如下：

```
mcr<cond> pn, opcode1, rd, crn, crm<, opcode2>  
mrc<cond> pn, opcode1, rd, crn, crm<, opcode2>
```

它的格式与 **cdp** 指令类似，只不过目标寄存器换成了 **ARM** 处理器自己的寄存器，其他部分的含义也与 **cdp** 指令类似。

如果协处理器不能成功执行指定的操作，就会产生未定义指令异常。

4.3.9 伪指令

为了编程方便，ARM 汇编中可以使用一些伪指令。这些伪指令并不与某条机器指令对应，而是由汇编器将其替换为一条或几条等效的 ARM 指令。

4.3.9.1 adr 伪指令

adr 伪指令用来向寄存器传送一个地址值，其格式如下：

```
adr<cond> rd, addr
```

其各个部分的含义解释如下。

- ◆ **cond**: 条件码助记符。
- ◆ **rd**: 用于存放地址的目标寄存器。
- ◆ **addr**: 一个地址值。

这里 **addr** 一般是程序中的一个标号，编译器产生的指令将把当前 **PC** 寄存器的值加上或减去一个固定偏移量作为地址值。

4.3.9.2 ldr 伪指令

由于使用 **mov** 或 **mvn** 指令向寄存器传送的立即数是受到一定限制的，因此有了 **ldr** 伪指令，它可以向寄存器传送任意的立即数，其格式如下：

```
ldr<cond> rd, =im32
```

其各个部分的含义解释如下。

- ◆ **cond**: 条件码助记符。
- ◆ **rd**: 目标寄存器。
- ◆ **im32**: 一个 32 位的常数。

它与存储器访问指令 **ldr** 的区别在于常数前面是一个等号。实际的实现可能是 **mov** 或 **mvn** 指令，也可能是将常数放在存储器中，用存储器访问指令 **ldr** 进行加载。

ldr 伪指令也可以用来传送地址，与 **adr** 伪指令的不同点在于，它不是以相对于当前 **PC** 寄存器值的方式来生成指令，因而不受偏移量的限制。

4.3.9.3 nop 伪指令

nop 伪指令不做任何操作，但是占用指令的执行时间，其格式如下：

```
nop
```

nop 伪指令可用来实现软件上的延时。

4.4 GNU ARM 汇编

不同的汇编器对汇编语言的语法要求有所不同。目前常用的 ARM 汇编环境有以下两种。

- ◆ **ARMASM**: ARM 公司的汇编器, 绝大多数 Windows 下的开发者都在使用这一环境, 它所采用的语法完全按照 ARM 的规定。
- ◆ **GNU ARM ASM**: GNU 交叉编译工具链中的汇编器, 所采用的语法与 ARMASM 略有不同, Linux 下的开发者一般使用这个环境。

本节将对 GNU ARM 汇编的语法进行介绍。

4.4.1 基本语法

在 GNU ARM 汇编中, 一行语句的基本格式如下:

标签: 指令 @ 注释

以冒号结尾的是一个标签, 而 @ 符号及后面的部分均被视为注释。在一行中, 标签、指令和注释三个部分均为可选的。

这里的指令不仅可以是 ARM 指令, 也可以是 GNU ARM 汇编所支持的一些指示性的关键字, 后者通常也被称为伪指令, 但要注意与 ARM 本身的伪指令概念上有所区别。

标签代表了标签处的代码或数据的地址, 可以用于分支指令 **b** 或 **bl**, 也可以用于 **ldr** 指令。一般来说, 整个文件中的标签名应该唯一, 但是 GNU 汇编器也支持一种局部标签的方式, 如:

```
        ldr    r1, =str
        mov    r2, r1
1:      ldr    r0, [r2], #1
        cmp    r0, #0
        beq    2f
        bne    1b
2:      sub    r3, r2, r1
```

局部标签只能是 0 ~ 99 的数字, 使用时要在数字后加上 **b** 或 **f**, **b** 表示向前搜索最近的一个标签作为目标, **f** 则表示向后搜索最近的一个标签作为目标。局部标签在宏定义中非常有用, 因为宏会在源码中多次使用, 也就意味着其内的标签会出现多次。

如果要注释整行, 则可以在行首加一个 **#** 字符。

另外, 指令中所使用的立即数前面应加一个 **#** 或 **\$** 字符。

一行中的多条语句可以用分号分隔。

下面是一个简单的汇编语言例程, 这段程序定义了一个 **add** 函数, 用于返回两个参数的和:

```
.text @ 代码段开始
.global add @ 声明 add 为全局标签, 这样它就可以在其他文件中使用
add: @ 标签
    add    r0, r0, r1 @ 将两个参数相加, 结果放在 r0 中
    mov    pc, lr @ 函数调用返回
```


其中用到了很多汇编伪指令，如 `.text` 和 `.global` 等。而指令的写法仍然是 ARM 指令的标准形式。因此下面将主要介绍一些常用的 GNU ARM 汇编伪指令。

4.4.2 GNU ARM 汇编伪指令

在进行汇编语言编程时，除了熟悉处理器所支持的各种汇编指令以外，最重要的一点就是掌握汇编环境所支持的一些伪指令。下面将对 GNU ARM 汇编中的常用伪指令进行简要的说明，这些伪指令的特点是都以一个小数点开头。

4.4.2.1 符号定义伪指令

`.equ` 伪指令用来定义一个常数，格式如下：

```
.equ symbol, value
```

其中 `value` 是一个值为常数的表达式。定义以后，`symbol` 这个符号就可以代替这个常数使用了。



GNU ARM 汇编也能识别 C 语言中的 `#define` 预编译指令。

`.globl` 伪指令用来声明全局标签，格式如下：

```
.globl label
```

其中 `label` 是程序中的一个标签。声明以后，`label` 将被导出到目标文件中，从而可以与其他目标文件链接在一起。也就是说，可以在其他源文件中使用 `label`。`.globl` 也可以写为 `.global`。

`.req` 伪指令用于定义寄存器的别名，格式如下：

```
alias .req reg
```

其中 `reg` 是一个寄存器，而 `alias` 是寄存器的别名。定义以后，这个别名就可以代替寄存器名在代码中使用。

4.4.2.2 数据定义伪指令

`.ltorg` 用于声明文字池，格式如下：

```
.ltorg
```

文字池的作用是存放 `ldr` 伪指令所传送的数据。文字池应该声明在程序正常执行到达不了的地方，如无条件跳转指令之后或子程序返回之后，以免其中的数据被当做指令来执行。如果不声明文字池，则汇编器会在程序的末尾自动加上文字池。

`.byte` 用于定义字节型数据，格式如下：

```
.byte byte1, byte2, ...
```

其中 `byte1`, `byte2` 等是字节型数据，可以定义多个。所定义的数据将放在从代码的当前位



置开始的一段存储空间内。

类似的还有定义半字（16 位）型和字（32 位）型数据的伪指令，格式分别如下：

```
.hword hword1, hword2, ...  
.word word1, word2, ...
```

其中 `.hword` 伪指令用于定义半字型数据，`hword1`、`hword2` 等都是 16 位数据，而 `.word` 伪指令用于定义字型数据，`word1`、`word2` 等都是 32 位数据。

`.ascii` 用于定义字符串，格式如下：

```
.ascii "string"
```

字符串要用双引号包围起来。所定义的字符串将放在从代码的当前位置开始的一段存储空间内。注意字符串的末尾并不会自动加上一个 `\0` 结束符。如果需要结束符，则可以用下面这条伪指令定义字符串：

```
.asciz "string"
```

`.asciz` 伪指令同样是定义字符串，并且会自动在字符串结尾多加一个 `\0`。

`.align` 伪指令用于填充一些字节以使随后的地址对齐指定的边界，格式如下：

```
.align power_of_2, fill_value
```

其各个部分的含义解释如下。

- ◆ `power_of_2`：随后的地址将是 2 的 `power_of_2` 次幂的整数倍。
- ◆ `fill_value`：要填充的字节，可选部分。

`.space` 伪指令用于填充空白数据，格式如下：

```
.space count, fill_byte
```

其中 `count` 是要填充的字节个数，`fill_byte` 则是要填充的字节。如果 `fill_byte` 省略，则默认填充 0。

4.4.2.3 汇编控制伪指令

一、代码分支

GNU ARM 汇编中支持如下形式的代码分支：

```
.if expr  
code1  
.else  
code2  
.endif
```

其中 `code1` 和 `code2` 代表两段汇编代码，而 `.if`、`.else` 和 `.endif` 都是汇编伪指令。`.if` 之后的 `expr` 则是一个表达式。整段代码的含义是：如果 `expr` 的值为真，则代码段 `code1` 将被处



理, `code2` 将被忽略; 否则如果 `expr` 的值为假, 则代码段 `code1` 将被忽略, 而 `code2` 将被处理。



GNU ARM 汇编也能识别 C 语言中的 `#if` 等预编译指令。与 `#if` 不同的是, `.if` 并不是在预编译阶段处理, 而是在汇编过程中处理。

二、循环

GNU ARM 汇编中支持如下形式的循环结构:

```
.rept times
code
.endr
```

其中 `code` 表示一段代码, `.rept` 和 `.endr` 都是汇编伪指令, `times` 是一个数字, 表示 `code` 这段代码要重复执行的次数。注意这里的所谓循环实际上是将 `code` 代码重复 `times` 次。

另一种循环的形式如下:

```
.irp param, val1, val2 ...
code
.endr
```

其中 `.irp` 也是汇编伪指令。`param` 是一个参数名, 其后的 `val1`, `val2` 则是它的取值列表。`code` 这段代码中可以引用参数 `param`, 形式是参数名前面加一个反斜杠 `\`。整段代码的含义是重复 `code` 代码若干次, 其中对 `param` 参数的引用依次替换为参数列表中的值。需要注意的是, 这种替换仅仅是一种简单的字符串替换, 如:

```
.irp n, 0, 1, 2
mov r\n, #0
.endr
```

这里参数仅仅是寄存器名的一部分, 它本身并不需要有完整的意义。

三、宏定义

GNU ARM 汇编中支持宏定义, 形式如下:

```
.macro name arg1, arg2, ...
code
.endm
```

其中 `.macro` 与 `.endm` 是汇编伪指令, `name` 则是宏的名称, 其后的 `arg1`, `arg2` 则是宏的参数列表。在 `code` 这段代码中可以使用这些参数, 使用的方法也是在参数名前加反斜杠 `\`。

宏定义本身并不生成目标代码, 但所定义的宏可以在随后使用, 相当于将 `code` 代码重写一遍并将引用的参数替换成使用时提供的对应值。这种替换也是一种简单的字符串替换。

在一个宏内部还可以使用 `.exitm` 伪指令跳出一个宏, 举例如下:

```
.macro shift_left a, b
.if \b < 0 @ 如果 b 是负数
```

```
mov \a, \a, asr #-\b @ 寄存器 a 算术右移 -b 位
.exitm @ 跳出宏
.endif
mov \a, \a, LSL #\b @ 否则寄存器 a 逻辑左移 b 位
.endm @ 结束宏
```

定义了这个宏以后，可以如下使用：

```
shift_left r0, -2
```

这个宏展开以后，实际上是如下指令：

```
mov r0, r0, asr #2
```

四、段定义

在 GNU ARM 汇编中用如下形式定义一个新段的开始：

```
.section name, "flags"
```

这里 `.section` 是汇编伪指令，`name` 是段的名称。`flags` 则是放在双引号中的若干个字符，表示段的访问权限，是可选部分，如果要生成 ELF 格式的可执行程序，则它有如下几个允许的取值。

- ◆ a：表示允许访问。
- ◆ w：表示允许写。
- ◆ x：表示允许执行。

汇编器能够识别一些预定义的段名，如：

- ◆ `.text`：表示代码段。
- ◆ `.data`：表示初始化的数据段。
- ◆ `.bss`：表示未初始化的数据段。

它们的访问标志是默认的，对于这些预定义的段名，使用时可以同时省略 `.section` 关键字和访问标志。这些预定义的段名已经写在链接器的默认链接脚本内。如果有自定义的段名，则必须提供链接脚本才能成功链接。

新段开始以后直到下一个新段定义之间的代码均属于这个段。

五、文件包含及其他

`.include` 伪指令用于包含其他文件的内容，如：

```
.include "filename"
```



C 语言中的 `#include` 预编译指令在 GNU ARM 汇编中也是可以识别的，只不过两者处理时所处的阶段不同，一个是在预编译阶段，一个是在汇编阶段。

由于 ARM 处理器可能同时支持 ARM 指令和 Thumb 指令，所以必须对汇编器有所指示，如：

```
.arm @ 指示随后的代码应编译为 ARM 指令
.code 32 @ 指示随后的代码应编译为 ARM 指令
.thumb @ 指示随后的代码应编译为 Thumb 指令
.code 16 @ 指示随后的代码应编译为 Thumb 指令
```

.err 伪指令则可以使编译过程出错并中止。

.end 伪指令表示汇编源代码结束，通常可以省略。

4.5 汇编与 C 语言

指令实际是处理器提供的最底层的软件接口，处理器的基本功能就是执行指令，而指令序列构成了程序最重要的代码部分。程序代码本身也可以放在存储器中，可以说，把程序放在存储器中是现代计算机发展史上最重要的里程碑。

程序的开发一般使用高级语言（如 C 语言）进行，所写的源程序必须经过编译器的处理转化为汇编语言，并进一步转化为机器指令才能被处理器识别。在嵌入式开发中，往往需要深入到系统内部进行优化，进行 C 语言和汇编语言的混合编程，这都离不开对两者之间关系的深入了解。

这一节将主要围绕一些实例探讨 C 语言与汇编语言的关系及编译器在其中扮演的角色，同时介绍了 ATPCS 约定的一些内容。

4.5.1 程序及其二进制映像

可执行的代码与被这些代码所使用的数据共同构成了一个程序。从处理器的角度看，指令是执行的基本单元，处理器只负责忠实地按照既定的指令进行执行，执行的可靠性是由硬件保证的。而指令序列是否合理则是软件设计者所要做的工作。为了便于加载与管理程序以及构筑大型的程序，现代的操作系統对程序在内存中的存放是有结构的，一般来说，分为以下几个部分，每个部分在内存中是连续存放的。

- ◆ 代码段：这部分是程序的可执行代码，由操作系统从可执行文件中加载。
- ◆ 数据段：这部分是程序所使用的全局数据，由操作系统设置并从可执行文件中加载其初值。
- ◆ 堆：现代的操作系統一般都支持程序动态分配内存，所分配的内存放在这里，这部分的大小是可变的。堆由操作系统自动设置。
- ◆ 栈：为了支持函数调用机制，必须设置一个栈以存放函数的局部变量、返回地址等信息，栈的大小一般也是可变的。栈由操作系统自动设置。

程序的各个部分在内存中的布局如图 4.7 所示。

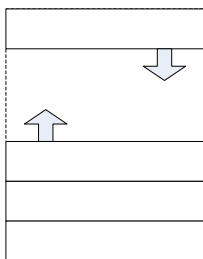


图 4.7 程序在内存中的布局

必须说明的是,这是当程序在操作系统上运行时的内存布局。如果程序不是在操作系统上运行,则不一定有这些结构。当用汇编语言开发程序时,我们可以人为地设置代码段与数据段以满足操作系统的要求。当用 C 语言开发程序时,这些底层的细节被掩盖了,实际上是由编译器自动进行设置的,比如,全局变量被放在数据段上,函数代码被放在代码段上,而局部变量和参数等实际上是在栈上动态生成的。

4.5.2 程序的编译与运行

源代码必须变成二进制的可执行代码后,才能被系统加载执行,这个过程中编译器担当了重要的角色。实际上,整个编译的过程是由好几个独立的步骤组成的。

C 语言源程序首先被编译为汇编语言程序,然后由汇编器转化为机器语言的目标文件。多个目标文件可由链接器进行链接,组合成一个可执行程序。运行时,操作系统的加载器读取这个可执行程序,放入内存中的合适位置以便处理器执行。下面将具体说明各个步骤所进行的处理。

一、编译

编译器对 C 程序进行解析,转换成汇编语言程序。

二、汇编

汇编指令实质上只是机器指令的助记符,必须经过汇编器的处理,转化为二进制形式的机器指令后才可以被处理器执行。转化以后的机器代码形成一个目标文件,但还不是可执行文件。

为了产生汇编语言程序中每条指令对应的二进制表示,汇编器必须处理所有标签对应的地址。汇编器将代码中用到的标签都放入一个符号表中,这个表的每一项包含了符号和对应的地址。

Linux 系统中的目标文件通常包含以下部分。

- ◆ 首部: 主要描述目标文件其他部分的大小和位置。
- ◆ 正文段: 包含机器语言代码。
- ◆ 数据段: 包含各种变量的初值。
- ◆ 重定位信息: 记录了模块重定位时需要修改的地址值及修改的位置。
- ◆ 符号表: 主要保存各个标签的相关信息,如与地址的对应关系,也包括引用的外部符号。
- ◆ 调试信息: 供调试器使用的辅助信息。

三、链接

链接器将单独编译得到的所有目标文件链接在一起,为所有的指令与数据统一编址,其工作主要有如下三个步骤。

- step 1** 将代码和数据模块重新进行排列组合,即重定位。
- step 2** 确定数据与指令中的地址。
- step 3** 修正内部或外部引用。

链接器使用每个目标模块中的重定位信息和符号表,来解析所有未定义的标签。这种引用发生在分支指令或数据寻址处,所有的旧地址被重定位后的新地址取代。

链接器将产生一个可执行文件,这个文件与目标文件具有相同的格式,但它已解决了所有的地

址引用问题，是可以载入内存进行执行的文件。一个例外是使用了共享库，对共享库中的标签的引用将在程序运行时解决。

四、加载

操作系统将可执行文件读到内存中并启动运行。在 Linux 系统中，这个过程的主要操作如下。

- ◆ 读取可执行文件首部以便确定正文段和数据段的大小。
- ◆ 为正文和数据创建一个足够大的地址空间。
- ◆ 把可执行文件中的指令和数据复制到相应的内存中。
- ◆ 把主程序的参数（如果存在）复制到栈的顶部。
- ◆ 初始化机器寄存器，将栈指针指向第一个空单元。
- ◆ 跳转到启动例程，它将参数赋值到参数寄存器并且调用程序的主函数。

4.5.3 ATPCS 约定

在汇编语言与 C 语言混合编程的情况下，或者是各个源文件不是由同一个编译器进行编译的情况下，所有汇编代码的生成（或编写）都必须遵循统一的调用接口规范，这样才能保证链接出来的程序功能是正确的。

ATPCS（ARM/Thumb Procedure Call Standard，ARM/Thumb 过程调用标准）是 ARM 公司定义的汇编层次上的统一的调用接口标准。编译器一般都遵循这个标准，因此，如果要在操作系统中进行汇编语言级的开发，就必须了解 ATPCS 标准，因为系统的各种函数接口都是以 C 语言的形式提供的。如果所写的汇编代码并不直接与 C 语言代码相互调用，可以不遵循这个规范。

ATPCS 约定具体有如下方面。

- ◆ 寄存器使用规定。
- ◆ 栈使用惯例。
- ◆ 栈回溯结构的格式。
- ◆ 参数传递与结果返回规则。
- ◆ ARM 共享库机制。

4.5.3.1 寄存器使用规定

ATPCS 对通用寄存器的用途做了一些规定，并且根据不同用途为寄存器定义了别名，如表 4.8 所示。

表 4.8 通用寄存器的使用与别名

寄存器	别名	用途
r0	a1	函数参数和返回值
r1	a2	函数参数和返回值
r2	a3	函数参数
r3	a4	函数参数
r4	v1	变量
r5	v2	变量



r6	v3	变量
(续表)		
寄存器	别名	用途
r7	v4	变量
r8	v5	变量
r9	v6, sb	变量, 或静态数据基址
r10	v7, sl	变量, 或栈限制
r11	v8, fp	变量, 或帧指针
r12	ip	函数调用中间临时寄存器
r13	sp	栈指针
r14	lr	链接寄存器
r15	pc	程序计数器

r0 - r3 这 4 个寄存器用来传递函数调用的第 1 到第 4 个参数, 更多的参数则必须通过栈来传递。而 r0 同时用来存放函数调用的返回值。被调用的子程序在返回前无须恢复这些寄存器的内容。

r4 - r11 这 8 个寄存器可作为一般的临时变量使用, 子程序进入时必须保存这些寄存器的值, 在返回前必须恢复这些寄存器的值。其中 r9 往往用来存储全局变量的基址, 而 r10 用来存储对栈的限制, r11 则作为指针指向函数调用帧。

r12 在函数调用时扮演着临时保存栈指针的角色, 而 r13 则作为栈指针, r14 是链接寄存器, r15 是程序计数器, 这些寄存器最好不要被移作他用。

4.5.3.2 栈使用惯例

首先总结一下栈的概念: 栈是一块存储区域, 其存储和读取数据的位置由栈指针控制。当数据被存入时, 栈指针的值被修改使栈的长度增大, 称为数据入栈; 当数据被取出时, 栈指针的值被修改使栈的长度减小, 称为数据出栈。数据的入栈与出栈遵循后进先出的原则。

栈可以向内存的低地址方向生长, 也可以向高地址方向生长, 分别被称为递减型和递增型。另外栈指针可以在数据入栈前修改, 也可以在入栈后修改。前者被称为满栈, 因为栈指针指向的位置总是有数据的; 而后者被称为空栈, 因为栈指针指向的位置没有数据。这样相互组合共有 4 种类型的栈, 如图 4.8 所示。

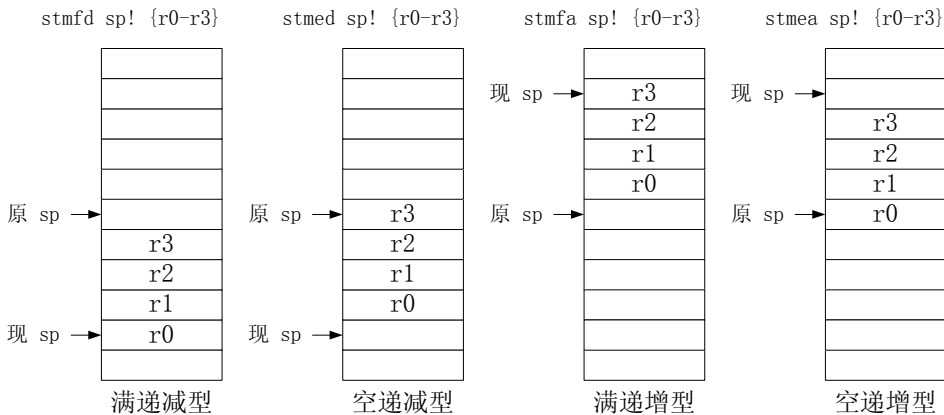


图 4.8 栈的类型

ATPCS 约定使用满递减类型的栈。

4.5.3.3 参数传递与结果返回规则

调用函数时，当参数不超过 4 个时，使用寄存器 r0 - r3 来传递，多于 4 个的参数要用栈来传递，入栈的顺序是从后往前，即最后一个参数先入栈。

如果参数类型的长度大于一个字，则将其视为连续的多个字数据进行处理。

如果函数的返回值为 32 位及以下长度的整数，则可以放在 r0 中返回；如果函数的返回值为 64 位的整数，则放在 r0 和 r1 中返回。对于位数更多的返回值，实际上是通过传入一个指针参数来返回的。

如果参数或返回值是浮点数，则有可能使用浮点协处理器的寄存器返回。

4.5.4 汇编与 C 语言的对照

在这里，我们将对简单的 C 语言源程序和编译后生成的汇编代码进行对照，以加深读者对两者之间关系的理解。需要指出的是，不同版本的编译器，或者同一编译器使用不同的编译参数，产生的汇编代码也是不同的。为了更好地进行代码的对照，编译时不能打开编译器的优化选项。

4.5.4.1 变量和算术运算

C 语言中的算术运算大多数可以直接用 ARM 指令来完成。比如一个简单的 C 语言程序如下：

```
/* 文件名: test1.c */
/* 说明: 变量与算术运算 */

int v1 = 1;
static int v2 = 2;

int main(void)
{
    int vr;
    int v3 = 3, v4 = 4;
    vr = (v1 + v2) - (v3 + v4);
    return vr;
}
```

使用如下命令编译，将生成汇编文件 test1.s:

```
arm-linux-gcc -S test1.c
```

test1.s 文件的内容如下（增加了注释）:

```
.file "test1.c"
.global v1 @ 声明 v1 为全局标签
.data @ 数据段开始
.align 2 @ 地址与 4 的倍数对齐
.type v1, %object @ v1 标签代表数据对象
```

```

        .size    v1, 4 @ 对象 v1 的长度为 4
v1:
        .word    1 @ 存放 v1 的初始值 1
        .align   2 @ 地址与 4 的倍数对齐
        .type    v2, %object @ v2 标签代表数据对象
        .size    v2, 4 @ 对象 v2 的长度为 4
v2:
        .word    2 @ 存放 v2 的初始值 1
        .text    @ 代码段开始
        .align   2
        .global  main @ 声明 main 为全局标签
        .type    main, %function @ main 标签代表函数
main:
        @ args = 0, pretend = 0, frame = 12
        @ frame_needed = 1, uses_anonymous_args = 0
        mov     ip, sp @ 暂时用 ip 保存栈指针
        stmfd   sp!, {fp, ip, lr, pc} @ 入栈
        sub     fp, ip, #4 @ fp 指向入栈的第一个元素 (帧的开始)
        sub     sp, sp, #12 @ 在栈上开辟 3 个整型数的空间, 用于存放局部变量
        mov     r3, #3
        str     r3, [fp, #-20] @ 将变量 v3 赋初值为 3
        mov     r3, #4
        str     r3, [fp, #-24] @ 将变量 v4 赋初值为 4
        ldr     r3, .L2 @ 将变量 v1 的地址加载到 r3
        ldr     r2, .L2+4 @ 将变量 v2 的地址加载到 r2
        ldr     r1, [r3, #0] @ 将变量 v1 的值加载到 r1
        ldr     r3, [r2, #0] @ 将变量 v2 的值加载到 r3
        add     r1, r1, r3 @ v1 + v2 的结果放入 r1
        ldr     r2, [fp, #-20] @ 将变量 v3 的值加载到 r2
        ldr     r3, [fp, #-24] @ 将变量 v4 的值加载到 r3
        add     r3, r2, r3 @ v3 + v4 的结果放入 r3
        rsb     r3, r3, r1 @ (v1+v2) - (v3+v4), 结果放入 r3
        str     r3, [fp, #-16] @ 结果保存到 vr
        ldr     r3, [fp, #-16] @ 将 vr 的值加载到 r3
        mov     r0, r3 @ 设置返回值
        sub     sp, fp, #12 @ 重设栈指针, 准备返回
        ldmfd   sp, {fp, sp, pc} @ 返回
.L3:
        .align   2
.L2:
        .word    v1 @ 变量 v1 的地址, 即标号 v1
        .word    v2 @ 变量 v2 的地址, 即标号 v2
        .size    main, .-main @ main 函数的大小, 当前位置减去 main 标号处
        .ident   "GCC: (GNU) 3.4.4"

```

可以看到, 因为每条指令只能执行一个简单运算, 所以编译器必须将 C 语言的一个复杂表达式翻译成多条汇编语言指令, 计算的中间结果使用寄存器来保存。如果表达式非常复杂以至于无法用寄存器保存所有的中间结果, 则还会在栈上开辟局部变量来保存。

全局变量都被放在数据段上, 数据段中保存的其实是变量的初始值。未用 `static` 声明的全局

变量被声明为 `.global`，表示它可以链接到其他文件。加载全局变量实际上使用了文字池的方法，即将变量地址存放在代码段中某个不会执行到的位置，使用时先加载变量地址，然后通过变量地址得到变量的值。

非静态的局部变量都放在栈上，通过帧指针加偏移量的方式来访问。帧指针在开始时设好，在整个函数执行期间不会改变。

4.5.4.2 访问数组

C 源程序的内容如下：

```
/* 文件名: test2.c */
/* 说明: 数组访问 */

int v1 = 5;
int A[10] = {0, 1, 2};

int main(void)
{
    int vr;
    int i = 2;
    vr = A[i];
    return vr;
}
```

编译生成的汇编代码如下：

```
.file "test2.c"
.global v1
.data @ 数据段开始
.align 2
.type v1, %object
.size v1, 4
v1:
.word 5
.global A
.align 2
.type A, %object
.size A, 40 @ A 的大小为 40 字节 (10 个字)
A:
.word 0 @ A[0] 的初值为 0
.word 1 @ A[1] 的初值为 1
.word 2 @ A[2] 的初值为 2
.space 28 @ 随后的元素未初始化, 被填充为 0
.text @ 代码段开始
.align 2
.global main
.type main, %function
main:
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
```



```

mov    ip, sp
stmfd  sp!, {fp, ip, lr, pc}
sub    fp, ip, #4
sub    sp, sp, #8 @ 在栈上为 vr 和 i 预留空间
mov    r3, #2
str    r3, [fp, #-20] @ 将 i 赋值为 2
ldr    r2, .L2 @ 加载数组 A 的首地址
ldr    r3, [fp, #-20] @ 加载 i 的值
ldr    r3, [r2, r3, asl #2] @ 以数组首地址为基址, i 为索引加载数据
str    r3, [fp, #-16] @ 结果保存到变量 vr
ldr    r3, [fp, #-16]
mov    r0, r3 @ 设置返回值
sub    sp, fp, #12 @ 重设栈指针, 准备返回
ldmfd  sp, {fp, sp, pc} @ 返回
.L3:
    .align 2
.L2:
    .word A
    .size main, .-main
    .ident "GCC: (GNU) 3.4.4"

```

可以看到,编译器处理数组的方式与一般变量是一致的,只不过数组会连续占用多个存储空间。

值得一提的是,在访问数组元素时,它的地址是由数组的首地址加上下标乘以数组元素的大小而得到的。由于 ARM 指令的第二操作数有移位功能,当数组元素的大小为 2 的整数次幂时,使用一条指令即可同时完成地址计算和数据加载,这对数组的访问来说是十分高效的。

4.5.4.3 分支结构

为了节省篇幅,我们这里只给出了 C 源程序的片段:

```

if (a == 0) {
    b = 1;
} else {
    b = 2;
}

```

其中的变量 a 和 b 都被定义为局部变量。编译后生成的汇编代码(对应片段)如下:

```

ldr    r3, [fp, #-16] @ 读变量 a 的值
cmp    r3, #0 @ 与 0 比较
bne    .L2 @ 如果不等则跳转到标签 .L2 处
mov    r3, #1
str    r3, [fp, #-20] @ 给变量 b 赋值为 1
b      .L1 @ 跳转到标签 .L1 处
.L2:
mov    r3, #2
str    r3, [fp, #-20] @ 给变量 b 赋值为 2
.L1:

```

通过上述例子我们看到,在高级程序语言中,一般并不出现跳转语句,但最终是由编译器用汇



编的分支指令实现的。

由于我们没有打开编译器的优化选项，因此这段汇编代码是忠实地按照 C 语言的流程生成的，执行效率很低。实际上，由于 ARM 的每条指令都有一个条件码，上述代码可以大大简化以提高效率，如：

```
ldr    r3, [fp, #-16] @ 读变量 a 的值
cmp    r3, #0 @ 与 0 比较
moveq  r3, #1 @ 如果相等则 b 为 1
movne  r3, #2 @ 如果不等则 b 为 2
str    r3, [fp, #-20] @ 保存变量 b 的值
```

这样写不仅代码简洁，而且能够最大程度地避免由于条件分支带来的处理器流水线重新装载，能极大地提高程序的执行效率。

4.5.4.4 循环结构

C 源程序的内容如下：

```
/* 文件名: test4.c */
/* 说明: 循环结构 */

int main(void)
{
    int i, sum;
    int A[] = {1, 2, 3, 4, 5};
    for (i = 0; i < sizeof(A)/sizeof(int); i++) {
        sum += A[i];
    }
    return sum;
}
```

生成的汇编代码如下：

```
.file "test4.c"
.section .rodata @ 开始一个只读数据段
.align 2
.LC0: @ 这里存放数组的初始化数据
.word 1
.word 2
.word 3
.word 4
.word 5
.text @ 代码段开始
.align 2
.global main
.type main, %function
main:
    @ args = 0, pretend = 0, frame = 28
    @ frame_needed = 1, uses_anonymous_args = 0
    mov ip, sp
```

```

stmfd    sp!, {fp, ip, lr, pc}
sub      fp, ip, #4
sub      sp, sp, #28 @ 为变量 i, sum 和数组 A 预留空间
ldr      r3, .L5 @ 得到数组初始化数据的首地址
sub      lr, fp, #40 @ lr 指向数组首地址
mov      ip, r3 @ ip 指向初始化数据的首地址
# 以下四句为对数组 A 的初始化
ldmia    ip!, {r0, r1, r2, r3}
stmia    lr!, {r0, r1, r2, r3}
ldr      r3, [ip, #0]
str      r3, [lr, #0]
mov      r3, #0
str      r3, [fp, #-16] @ 将 i 赋值为 0
.L2:
ldr      r3, [fp, #-16] @ 读 i 的值
cmp      r3, #4 @ 与 4 相比
bhi      .L3 @ 如果大于 4 则跳转到循环结束处
ldr      r3, [fp, #-16] @ 读 i 的值
mvn      r2, #27 @ 让 r2 等于 -28
mov      r3, r3, asl #2 @ 将 i 乘以 4
sub      r1, fp, #12 @ 取得局部变量的开始地址
add      r3, r3, r1 @ r3 = fp-12+i*4
add      r3, r3, r2 @ r3 = fp-12-28+i*4, 数组 A 的首地址
ldr      r2, [fp, #-20] @ 读 sum 的值
ldr      r3, [r3, #0] @ 读 A[i] 的值
add      r3, r2, r3 @ sum + A[i], 放入 r3
str      r3, [fp, #-20] @ 将 r3 的值保存回 sum
ldr      r3, [fp, #-16] @ 读 i 的值
add      r3, r3, #1 @ i 增加 1
str      r3, [fp, #-16] @ 保存 i 的值
b        .L2 @ 跳回 .L2 处
.L3:
ldr      r3, [fp, #-20]
mov      r0, r3
sub      sp, fp, #12
ldmfd    sp, {fp, sp, pc}
.L6:
.align   2
.L5:
.word    .LC0
.size    main, .-main
.ident   "GCC: (GNU) 3.4.4"

```

这里我们看到，编译器仍是使用判断与分支指令来实现循环的再次开始和终止的，这与在 C 语言中用 `goto` 语句实现循环是一个道理。

另外一个现象是编译器对于局部数组的处理，这时数组元素是保存在栈上的，但数组的初始化数据被放在一个只读数据段上，当数组生成时使用指令对数组元素一一进行赋值。

4.5.5 函数调用与栈

函数是模块化的代码序列，是实现程序模块化的机制。使用函数有助于提高程序的可读性与可重用性。这不仅仅是程序设计上的一个逻辑概念，也与处理器结构的设计相关。

实现函数的基础是栈的使用。函数中使用的局部变量、调用者传递的参数都是在栈上分配的，函数嵌套调用的实现也是基于栈的。栈使用的一个重要原则就是：当一个函数调用返回后，栈要恢复到调用前的状态。另外，除了用于保存返回值的寄存器，调用者使用的其他寄存器都要恢复到函数调用前的值。

下面我们仍然通过 C 语言与汇编代码对照的方式来理解函数调用的实现。所用的 C 源代码如下：

```
/* 文件名: test5.c */
/* 说明: 函数调用 */

int foo(int a1, int a2, int a3)
{
    return a1 + a2 + a3;
}

int main(void)
{
    int v1 = 1, v2 = 2, v3 = 3;
    return foo(v1, v2, v3);
}
```

编译生成的汇编代码如下：

```
.file      "test5.c"
.text
.align    2
.global   foo
.type     foo, %function
foo:
    @ args = 0, pretend = 0, frame = 12
    @ frame_needed = 1, uses_anonymous_args = 0
    mov     ip, sp
    stmfd   sp!, {fp, ip, lr, pc}
    sub     fp, ip, #4
    sub     sp, sp, #12 @ 为参数开辟局部变量
    str     r0, [fp, #-16] @ 保存参数 a1
    str     r1, [fp, #-20] @ 保存参数 a2
    str     r2, [fp, #-24] @ 保存参数 a3
    ldr     r2, [fp, #-16] @ 读参数 a1 的值
    ldr     r3, [fp, #-20] @ 读参数 a2 的值
    add     r3, r2, r3 @ r3 = a1 + a2
    ldr     r2, [fp, #-24] @ 读参数 a3 的值
    add     r3, r3, r2 @ r3 = r3 + a3
    mov     r0, r3 @ 设置返回值
    sub     sp, fp, #12 @ 重设栈指针, 准备返回
    ldmdfd  sp, {fp, sp, pc} @ 返回
.size     foo, .-foo
```

```

.align 2
.global main
.type main, %function
main:
    @ args = 0, pretend = 0, frame = 12
    @ frame_needed = 1, uses_anonymous_args = 0
    mov     ip, sp
    stmfd   sp!, {fp, ip, lr, pc}
    sub     fp, ip, #4
    sub     sp, sp, #12
    mov     r3, #1
    str     r3, [fp, #-16] @ 变量 v1 赋初值为 1
    mov     r3, #2
    str     r3, [fp, #-20] @ 变量 v2 赋初值为 2
    mov     r3, #3
    str     r3, [fp, #-24] @ 变量 v3 赋初值为 3
    ldr     r0, [fp, #-16] @ 读 v1 的值到 r0
    ldr     r1, [fp, #-20] @ 读 v2 的值到 r1
    ldr     r2, [fp, #-24] @ 读 v3 的值到 r2
    bl      foo @ 调用 foo
    mov     r3, r0 @ 通过 r0 得到返回值
    mov     r0, r3
    sub     sp, fp, #12
    ldmdfd  sp, {fp, sp, pc}
    .size   main, .-main
    .ident  "GCC: (GNU) 3.4.4"

```

可以看出，编译器生成的代码是符合 ATPCS 约定的。调用函数前，三个参数分别放到寄存器 r0, r1, r2 中。在函数中，参数被作为局部变量处理，保存在栈上，随后又从栈中取出进行运算，并将结果放在 r0 中返回。

理解的关键是函数调用发生时，栈及栈中数据的变化情况。如图 4.9 所示是当函数调用发生时，栈初始化完毕后各寄存器指针的指向及栈中内容的示意。

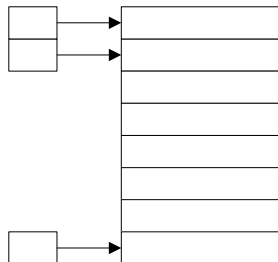


图 4.9 函数的调用栈

函数的复杂性在于需要在栈中分配函数的局部变量，包括函数的输入参数。当然，经过优化的代码有可能使用寄存器来存放局部变量。函数在栈中保存的所有局部信息，包括返回地址、局部变量等构成了函数的帧。

ATPCS 约定使用 fp 寄存器作为帧指针，指向函数帧的第一个字。在函数执行过程中栈指针

是可能改变的，因此用一个稳定不变的帧指针来引用局部变量是一种合理的做法。

帧指针 `fp` 指向当前函数帧的第一个字，而帧内保存的 `fp` 指针是调用者的 `fp` 指针，这样就构成了所谓的回溯结构（`backtrace structure`），也就是说，一旦程序运行中有问题发生，可以通过栈上的回溯结构来追踪函数的调用层次。

4.5.6 堆的概念

与栈相对应的一个概念是堆，很多时候直接将栈称为堆栈，但实际上，堆是有确切含义的、不同于栈的一个概念。

堆对应于动态内存分配的存储区域，堆的大小和位置由系统加载程序时确定，并且它的容量可以动态增加。在 C 语言程序中用 `malloc` 函数得到的内存就位于堆中，堆中内存的管理是由操作系统来负责的。如果开发的程序在没有操作系统的环境中运行，这时也常常会自己开发一个类似 `malloc` 的函数进行内存分配，也就是说，对内存的管理必须由开发者自己实现。

4.6 汇编与 C 语言混合编程

我们知道 C 语言是一个系统级的语言，也就是一个能够直接深入到硬件最底层操作的语言。对硬件的操作实际上是对硬件控制器中的寄存器或存储单元进行操作，而在 ARM 架构中，这些寄存器或存储单元一般是以内存映射的方式进行访问的。在 C 语言里可以使用指针的方式去访问任意的内存地址，因此对硬件的操作从语言上是可以表达的。

但是在操作底层硬件时，C 语言还是有其局限性的。有些硬件是没有地址一说的，比如处理器的寄存器、协处理器和协处理器的寄存器、系统控制器等，这些硬件资源是不可能使用 C 语言来访问的，只能用汇编指令。

因此，虽然嵌入式开发中大多数情况下我们倾向于使用 C 语言，但是在少数情况下还必须使用汇编语言，这就提出了 C 语言与汇编语言混合编程的要求。

对于 C 语言与汇编语言混合使用的方式主要有如下几种情况。

- ◆ 汇编程序中调用 C 语言函数。
- ◆ 汇编程序中使用 C 程序中定义的全局变量。
- ◆ C 语言中调用汇编语言的函数。
- ◆ C 语言中使用汇编程序中定义的全局变量。
- ◆ C 语言中内嵌汇编指令。

特别要注意最后一种方式，这是一种常见的方式，但它的实现与编译器有关，各种不同的编译器有其特有的内嵌汇编的形式，本节只介绍 GNU GCC 中的内嵌汇编形式。

4.6.1 C 语言调用汇编函数

这个例程中用汇编实现了一个字符串复制的函数，然后在 C 源程序中调用它。

C 语言源程序如下：

```
/* 文件名: string_copy.c */
```



```
/* 说明：字符串复制，C 调用汇编 */

#include <stdio.h>

/* 声明外部函数 */
extern char *asm_strcpy(char *dst, const char *src);

int main(void)
{
    char *src = "asm_strcpy test!";
    char dst[64];
    asm_strcpy(dst, src); /* 调用汇编函数 */
    printf("dst: %s\n", dst);
    return 0;
}
```

其中所调用的 `asm_strcpy` 函数实现在以下汇编源文件中：

```
# 文件名：asm_strcpy.S
# 说明：字符串复制，C 调用汇编

        .globl  asm_strcpy
        .text
asm_strcpy:
    mov     r4, r0 @ 保存 r0 的值
loop:
    ldrb     r3, [r1], #1 @ 从源字符串读一个字节
    strb     r3, [r0], #1 @ 向目标字符串写一个字节
    cmp     r3, #0 @ 判断字节是否为 0
    bne     loop
end:
    mov     r0, r4 @ 设置返回值
    mov     pc, lr @ 返回
```

使用以下命令即可将这两个文件编译并链接成可执行文件：

```
arm-linux-gcc string_copy.c asm_strcpy.S -o str_copy
```

这条命令将编译生成一个 `str_copy` 文件，将其下载到 ARM 目标机文件系统中，在 Shell 环境下输入文件名即可运行。

C 语言中调用汇编函数有以下几个要点。

- ◆ C 语言中用 `extern` 将用到的函数声明为外部函数。
- ◆ 汇编中用 `.globl` 或 `.global` 将函数的标号声明为全局类型。
- ◆ 汇编函数的写法需遵循 ATPCS 约定。

4.6.2 汇编语言中使用 C 全局变量

这个例程中在汇编程序内实现了对 C 语言的某个全局变量的求平方运算。



C 语言源程序如下：

```
/* 文件名: square.c */
/* 说明: 求平方, 汇编使用 C 全局变量 */

#include <stdio.h>

int var; /* 这个变量的值在汇编中被修改 */

extern void asm_square();

int main(void)
{
    var = 2;
    asm_square();
    printf("var = %d\n", var);
    return 0;
}
```

汇编源程序如下：

```
@ 文件名: asm_square.S
@ 说明: 求平方, 汇编使用 C 全局变量
.text
.globl asm_square
.extern var @ 可以不做这个声明
asm_square:
    ldr    r0, =var @ 得到变量 var 的地址
    ldr    r1, [r0] @ 得到 var 的值
    mul    r2, r1, r1 @ 乘法
    str    r2, [r0] @ 将结果保存回 var
    mov    pc, lr @ 返回
.ltorg @ 声明文字池, 可省略
.end
```

注意在汇编语言中甚至可以不做任何声明就使用 C 程序中的全局变量，一般建议加上一个 `.extern` 声明以增强可读性。

4.6.3 内嵌汇编

如果只是在 C 程序中有一小段需要用汇编实现的代码，那么单独写一个汇编源文件并进行相互调用就显得有些麻烦。这时一个有用的方式是在 C 程序中使用内嵌汇编。

GNU GCC 支持的 ARM 内嵌汇编格式如下：

```
asm("指令序列":输出列表:输入列表:修改内容);
```

其中 `asm` 是编译器可以识别的关键字。圆括号中的内容除了指令序列外，其他三个部分都可以省略。但如果省略了中间的部分，相应的冒号不可以省略。指令序列不能省略，但可以为空字符串。

内嵌汇编的使用举例如下：

```
asm("mov %[result], %[value], ror #1" : [result] "=r" (y) : [value] "r" (x));
```

这行代码的作用是将变量 `x` 循环右移 1 位并赋值给变量 `y`。
下面将对圆括号中冒号隔开的四个部分进行详细说明。

一、指令序列

指令序列以单一字符串的形式出现，其中包含要生成的汇编指令。在这些指令中可以使用输出列表和输入列表中定义的各种符号，如上述例子中的 `%[result]` 和 `%[value]`。这些符号在生成指令时将替换成对应的内容。

二、输出列表和输入列表

输出列表和输入列表的格式相同，都是由逗号隔开的多个部分组成的，代表多个输出操作数。每个部分由一个方括号包围的符号名、一个对操作数使用方式进行限定的字符串以及一个圆括号包围的 C 语言变量名组成。

在早期的 `gcc` 版本中，输出列表和输入列表中不支持符号名。这时指令序列中必须以 `%0`，`%1` 的方式进行引用，所用的数字是几就代表列表中的第几项（编号从 0 开始）。当然现在的 `gcc` 仍然支持这种方式。

输出列表和输入列表中常用的限定符如表 4.9 所示。

表 4.9 输出列表和输入列表中的限定符

限定符	含义
I	用做立即数
J	范围在 -4095 ~ 4095 内的常数
K	按位取反的立即数
L	取相反数的立即数
l	寄存器 r0 - r7
M	0 ~ 32 之间的常数或 2 的整数次幂
m	内存地址
r	寄存器 r0 - r15
X	任意用途

限定符前面还可以加一个修饰符进一步指明操作数的用途，不加任何修饰符表示是只读的，加一个等号表示是只写的，加一个加号表示既可读又可写。只读的操作数只能放在输入列表中，而只写和可读可写的操作数应放在输入列表中。

早期的 `gcc` 版本不支持加号修饰符，如果一个操作数既用于输出又用于输入，则只能在两个列表中同时出现，例如下面的代码：

```
asm volatile (  
    "eor r3, %1, %1, ror #16\n\t"  
    "bic r3, r3, #0xFF0000\n\t"  
    "mov %0, %1, ror #8\n\t"  
    "eor %0, %0, r3, lsr #8"  
    : "=r" (val)
```

```
: "0" (val)
: "r3");
```

这段代码用于交换一个 32 位整型变量 `val` 的字节序。限定符中出现的 0 表示这个操作数与第 0 个操作数是同一个变量。这种表示方式在现在的 `gcc` 编译器中仍然支持。在指令序列中插入了一些换行符和制表符，目的是为了让输出的汇编格式更美观。

三、修改列表

修改列表是一系列逗号隔开的字符串，代表在嵌入汇编中被修改过的内容。字符串的内容可以是一个寄存器名，代表这个寄存器的值被修改了，可以是 `cc`，表示条件标志被修改了。编译器将根据这些信息在内嵌汇编的前后进行保护处理，以免破坏 C 程序本身对寄存器的使用。

特殊一点的是 `memory`，表示要修改内存中的变量，这时编译器必须从内存中加载变量的值而不能用缓存在寄存器中的值，并且要在这条内嵌汇编语句结束后将变量的值写回内存。例如在 Linux 内核源码中经常用下面的内嵌汇编：

```
asm(":::"memory");
```

这就是所谓的“内存屏障”，编译器必须将它之前的变量写回内存，其后用到的变量必须再从内存中加载。

使用内嵌汇编需要注意的一点是，它仍然接受编译器的优化，甚至有可能被完全去掉。可以使用 `volatile` 关键字来防止这一点，如：

```
int temp;
asm volatile(
    "mrs %0, cpsr\n\t"
    "bic %0, %0, #0x80\n\t"
    "msr CPSR_c, %0"
    : "=r" (temp)
    :
    : "memory");
```

这段代码的作用是禁止中断。如果要在宏定义中使用内嵌汇编，则 `asm` 和 `volatile` 关键字可能引起编译器的警告，这时可用 `__asm__` 及 `__volatile__` 来代替，如上述用于交换字节序的内嵌汇编可定义为如下形式：

```
#define BYTESWAP(val) \
__asm__ __volatile__ ( \
    "eor r3, %1, %1, ror #16\n\t" \
    "bic r3, r3, #0x00FF0000\n\t" \
    "mov %0, %1, ror #8\n\t" \
    "eor %0, %0, r3, lsr #8" \
    : "=r" (val) \
    : "0"(val) \
    : "r3", "cc");
```

在不同的内嵌汇编代码中相互跳转是不允许的，但在同一个内嵌汇编中可以进行跳转。这时一般会使用局部标签，例如将前述字符串复制例程的主函数修改如下：



```
int main(void)
{
    char *src = "asm_strcpy test!";
    char dst[64];
    asm volatile (
        "1:\n\t"
        "ldrb r3, [%[src]], #1\n\t"
        "strb r3, [%[dst]], #1\n\t"
        "cmp r3, #0\n\t"
        "bne 1b\n\t"
        :
        : [src] "r" (src), [dst] "r" (dst)
        : "r3", "cc", "memory");
    printf("dst: %s\n", dst);
    return 0;
}
```



第 5 章 搭建嵌入式固件开发平台

嵌入式应用开发与主机环境下的开发相比，使用的软硬件工具更复杂一些，编译调试与运行软件也更难理解一些。而在非操作系统环境下的固件开发，更是如此，开发的程序常常缺乏如主机程序般方便的测试运行环境，不能及时看到效果，这常常是我们一些初学者望而却步的原因。很多希望进入嵌入式开发领域的人员，在固件低阶软件开发上都是因为找不到一个合适的学习开发与测试软件运行的环境，而止步于嵌入式开发的大门前的。

本章将介绍如何使用 `gcc` 交叉编译工具、`kermit` 终端以及开源的 `bootloader U-boot` 软件等搭建一个实用的固件开发测试平台，以帮助初学者更快地进入角色。需要强调的是，这些工具的功能并不逊于商业平台，是主流的、实用的开发方式。有了这样一个平台，就可以对固件程序进行开发和测试，以深入理解硬件控制器的工作机制，测试各种硬件接口。

5.1 硬件设备与软件环境

进行嵌入式 `Linux` 开发的主机必须是 `Linux` 操作系统，可以选择 `Debian 5.0` 发行版。个别工具软件需要在 `Windows` 环境下运行。如果只有一台计算机，可以考虑在 `Windows XP` 环境下搭配虚拟机来使用 `Debian 5.0`。

所需的具体硬件设备与软件环境如表 5.1 所示。

表 5.1 所需硬件设备与软件环境

硬件设备	HY2410A 开发板一块	
	JTAG 接口板及电缆（需要主机有并口）	
软件环境	RS232 串口电缆一条（需要主机有串口）	
	USB 到串口转换器（如果主机没有串口）	
	5V 电源	
	Windows 主机	Windows XP 操作系统 sjf2410 (S3C2410A 的 JTAG 烧写软件) 可以考虑使用虚拟机安装 <code>Linux</code> 超级终端或 <code>SecureCRT</code> （可代替 <code>Linux</code> 下的 <code>kermit</code> ）
Linux 主机		<code>Debian 5.0</code> 操作系统 <code>arm-linux-gcc 3.4.4</code> 交叉编译工具链 <code>kermit</code>

其中 `sjf2410` 工具仅用来为没有 `bootloader` 的开发板烧写 `bootloader`，如果不需要这一步操作，那么可以完全脱离 `Windows` 环境。

kermit 工具可以作为开发板的控制台，用来加载程序、运行程序以及监测运行结果。实际上 **kermit** 监测一个主机串口，这个串口与开发板的串口相连。通过这个串口可以将主机上交叉编译的程序下载到开发板内存，然后运行。而目标机的程序运行结果也通过这个串口发送回来，显示在 **kermit** 界面中。这样，通过 **kermit** 控制开发板与通过 **Shell** 控制主机是非常类似的。

当然，如果要达到这样的效果，还需要开发板上有一个运行环境，至少需要以下这些功能。

- ◆ 能够与主机进行通信，提供程序载入、信息输出等功能。
- ◆ 提供程序运行的环境，可以载入程序到指定内存，并能够执行程序。
- ◆ 能够将程序调试打印信息输送到串口。
- ◆ 最好提供一个交互式命令界面，提供各种有用的命令。

上述功能就是一个固件调测环境的基本功能，许多 **bootloader** 就具备了这些功能，如 **U-boot**。**U-boot** 是一个开源 **bootloader**，广泛应用在各种平台的嵌入式开发中。以其为基础，增加一些基本固件常用的 C 函数库，即可成为一个非常方便的固件开发平台。

5.2 搭建开发环境

本节将主要介绍 **HY2410A** 开发板与主机的硬件连线，及如何下载和运行我们开发的 **ARM** 二进制可执行程序。

5.2.1 硬件连接

如图 5.1 所示是开发板与主机的硬件连接，其中有三条主要的连接线，分别说明如下。

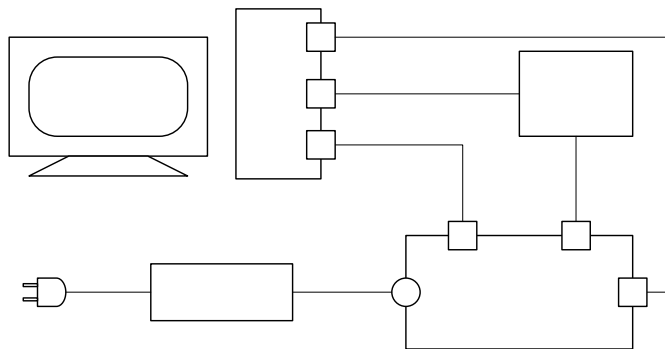


图 5.1 开发板与主机的硬件连接

- ◆ 串口线：这是一条 **RS232** 的串口直连线（开发板内部对串口的收发做了交叉）。如果主机没有串口，可以使用 **USB** 到串口的转换器。通过它可以对开发板进行控制，并接收开发板反馈的信息。
- ◆ 网线：这是一条交叉的五类线，用于直接连接主机和开发板。如果有更好的网络环境，也可以采用其他非直接连接的方式，只要保证主机与开发板的网络连接通畅即可。**HY2410A** 开发板支持 **10Mbit/s** 的以太网。网络的作用在于它下载文件到开发板的速度要远快于串口。它并不是必需的。

- ◆ JTAG 转接板：它将主机的并口转接到开发板的 JTAG 插座上。一般仅用它给裸板烧写 bootloader（因为没有 bootloader 的开发板无法通过上述方式下载文件），大多数情况下并不需要。

在这种开发模式下，常常把被控制、被调试的开发板（或其他嵌入式系统主机乃至 PC 机）称为目标机。

5.2.2 使用终端软件

硬件连接好之后下一步就是在主机上启动终端软件对串口进行监控，这里我们以运行于 Linux 系统上的 **kermit** 软件为例进行说明。首先启动 **kermit**，然后进行连接参数设置，对于 HY2410A 开发板来说，输入以下命令：

```
set line /dev/ttyS0 # 设置串口设备为 /dev/ttyS0
# 如果是 USB 转换的串口，则可能是 /dev/ttyUSB0
set serial 8N1 # 设置串口传输比特位为 8，校验位无，停止位为 1
set speed 115200 # 设置串口波特率为 115200b/s
set flow-control none # 设置流控制为无
set carrier-watch off # 设置载波监测为关闭
```

这些命令可以放入 **kermit** 的脚本中，这样就会在 **kermit** 启动后自动执行。参数设置好之后，用 **connect** 命令或 **c** 命令连接，连接成功将显示如下信息：

```
Connecting to /dev/ttyS0, speed 115200
Escape character: Ctrl-\ (ASCII 28, FS): enabled
Type the escape character followed by C to get back,
or followed by ? to see other options.
-----
```

连接成功之后，**kermit** 命令就不再起作用，因为这时界面上显示的是从串口读到的信息，而用户输入的信息将被发送到串口。一个例外是 **Ctrl+** 组合键，它不会被发送到串口，并且紧接着输入的一个字符将产生某种控制作用，也不会被发送到串口，比如输入字符 **?** 可以看到所有可能的字符的帮助信息，而输入字母 **C** 则会断开连接重新返回 **kermit** 界面。

连接成功之后给开发板加电，则会看到如下信息：

```
U-Boot 1.2.0 (Sep 17 2009 - 14:52:56)

DRAM: 64 MB
NAND: 64 MB
In: serial
Out: serial
Err: serial
Hit any key to stop autoboot: 0
HY2410>
```

这是 **U-boot** 输出的信息，最后的提示符表示已进入 **U-boot** 命令行界面，可以输入 **U-boot** 命令了。如果开发板已设置了自动启动命令，则 **U-boot** 启动后会有一个倒计时，在计时结束

前按任意键也将进入 U-boot 命令行界面。如果开发板尚未烧写 bootloader，则可能没有信息输出，这时必须用 JTAG 工具烧写 bootloader。

上述信息仅供参考，因为 U-boot 有很强的可定制性，因此不同的 U-boot 版本输出的信息可能有较大差别。

5.2.3 下载和执行程序

这里我们假设有一个编译好的可以直接执行的 ARM 二进制可执行文件 hello.bin，以它为例来说明在 U-boot 界面中下载和执行程序的过程。

5.2.3.1 通过串口下载文件

在 U-boot 界面中输入以下命令：

```
loadb 0x30000000
```

这条命令表示从串口下载文件，并放在从地址 0x30000000 开始的内存中。命令输入后将出现如下提示：

```
## Ready for binary (kermit) download to 0x30000000 at 115200 bps...
```

这表示 U-boot 在等待从串口接收文件。此时我们需要暂时离开 U-boot 界面回到 kermit 界面，即输入 Ctrl+\ 组合键，再输入 C。返回 kermit 界面后用以下命令发送文件：

```
send hello.bin
```

这条命令表示要发送 hello.bin 文件。开始发送后 kermit 将显示如下信息：

```
Current Directory: /home/jyg/work/arm
Communication Device: /dev/ttyS0
Communication Speed: 115200
    Parity: none
    RTT/Timeout: 01 / 01
    SENDING: hello.bin => HELLO.BIN
    File Type: BINARY
    File Size: 1012
    Percent Done: 100 //////////////////////////////////////////
                   ...10...20...30...40...50...60...70...80...90..100
    Elapsed Time: 00:00:01
    Transfer Rate, CPS: 708
    Window Slots: 1 of 1
    Packet Type: B
    Packet Count: 5
    Packet Length: 6
    Error Count: 0
    Last Error:
    Last Message: SUCCESS.  Files: 1, Bytes: 1012, 708 CPS
```

这些信息随传输的进度而动态更新。需要指出的是，串口传输本身是不太可靠的，因此在 kermit 默认的快速文件传送方式下一般不太可能传输成功，需要事先修改传送方式，这可以由以

下 `kermit` 命令做到：

```
robust
```

它实际上是 `kermit` 中定义的一些宏，包含了若干条设置传输方式的命令。传输完成后用命令 `c` 回到 `U-boot` 界面，可以看到输出的信息如下：

```
## Total Size      = 0x000000e8 = 232 Bytes
## Start Addr     = 0x30000000
```

其中显示了下载的字节数和所放置的内存地址。

5.2.3.2 通过网络下载文件

实际开发中往往通过网络来下载文件到目标机内存，因为它比用串口下载的速度快很多。这时首先要检查目标机的网络设置，即 `U-boot` 中的 `ethaddr`，`ipaddr`，`netmask` 三个环境变量，它们的值必须合理。

`U-boot` 使用 `TFTP` 协议来进行下载，因此要设置服务器地址，如：

```
setenv serverip 192.168.1.120
```

这里 `serverip` 环境变量的值就是服务器地址。然后可以用如下命令检查与服务器的通信是否正常：

```
ping $(serverip)
```

这里 `$()` 是 `U-boot` 中引用环境变量的方式。如果很快返回如下信息则说明通信正常：

```
host 192.168.1.120 is alive
```

接下来的步骤是确认主机的 `TFTP` 服务已在运行。把待下载的文件放入 `TFTP` 服务的共享目录，然后就可以在 `U-boot` 中下载文件了：

```
tftp 0x30000000 hello.bin
```

正常情况下将显示如下信息：

```
TFTP from server 192.168.1.120; our IP address is 192.168.1.128
Filename 'hello.bin'.
Load address: 0x30080000
Loading: #
done
Bytes transferred = 232 (e8 hex)
```

这说明下载成功了。

5.2.3.3 运行程序

由于我们的程序是可以直接运行的固件程序，现在它已经位于目标机内存中的某个地址处，所以直接用以下命令就可以运行：

```
go 0x30000000
```

这条命令表示跳转到地址 `0x30000000` 处执行。`go` 命令以函数的方式调用指定地址处的代码，所以如果程序以函数的形式返回的话就会回到 `U-boot` 界面。

5.3 创建固件程序

本节将通过一个简单的例程来说明开发 `ARM` 固件程序的基本方法。例程中包括以下两个源文件。

- ◆ `start.S`: 汇编语言源文件，用来跳转到 `C` 语言函数入口。
- ◆ `hello.c`: `C` 语言源文件，定义了一个 `C` 函数。

`start.S` 文件的源码如下：

```
/* 文件名: start.S */
/* 说明: 固件程序入口 */

.global _start
.text
_start:
    b      hello @ 调用 hello
.end
```

这个源码只有一句汇编语言，跳转到了函数 `hello`。再来看 `hello.c` 文件的源码：

```
/* 文件名: hello.c */
/* 说明: hello 例程 C 语言部分 */

void hello()
{
    printf("hello world!\n");
}
```

它的实现也很简单，在 `hello` 函数里面用 `printf` 函数输出了一个字符串。

这里有一个很重要的问题：我们的程序将在 `U-boot` 环境下执行，这时还没有操作系统，更没有 `C` 标准库提供各种 `API` 函数，那么所调用的 `printf` 函数从哪里来呢？

实际上，`U-boot` 本身实现了 `printf` 函数，它的用法与 `C` 标准库中的 `printf` 函数相似，作用则是向串口输出字符串，这样我们就可以在 `kermit` 环境中看到程序的运行结果。通过 `U-boot` 提供的一个静态库 `libstubs.a` 可以间接地调用到这些函数。

下面开始编译程序，首先将各个源码编译为目标文件：

```
arm-linux-gcc -c hello.c
arm-linux-gcc -c start.S
```

注意 `-c` 参数不能省略，否则编译器将试图像编译普通应用程序那样，寻找一个 `main` 函数并与应用程序的启动代码链接在一起，这显然是不行的。

然后将各个目标文件链接在一起：

```
arm-linux-ld -Ttext 0x30000000 -o hello start.o hello.o libstubs.a
```

这里我们只能直接用 `arm-linux-ld` 工具进行链接，不能用 `arm-linux-gcc`，因为后者仍会像处理普通应用程序那样来链接。

这里的参数 `-Ttext 0x30000000` 指定了 `text` 段从地址 `0x30000000` 开始。实际上，这个例程完全是位置无关的，在任何地址上都可以执行，因此可以不指定，链接器将为其指定一个默认的地址。如果程序中有绝对地址内容，则程序需要载入到指定的地址才能正常运行。

链接时 `start.o` 必须放在开始，因为它是整个程序的开始。

链接时同时加入了 `libstubs.a` 静态库，编译 `U-boot` 后可在它源码的 `examples` 目录中找到。

链接以后的文件是 `ELF` 格式的，并不能直接下载运行，需要将其转化为二进制格式：

```
arm-linux-objcopy -O binary -S hello hello.bin
```

其中 `-O binary` 表示生成的目标为二进制文件，`-S` 表示去掉原来文件中的所有符号和重定位信息，只留下纯粹的 `ARM` 指令。

这样得到的文件是可以直接运行的，将其下载到目标机内存的某个地址就可以执行了。注意因为 `U-boot` 本身的代码驻留在内存的高端，所以尽量不要把文件下载到内存的高端，以免破坏 `U-boot` 的代码，比如：

```
tftp 0x30000000 hello.bin
go 0x30000000
```

`0x30000000` 是 `HY2410A` 开发板内存的起始地址。

执行以后的结果如下：

```
## Starting application at 0x30000000 ...
hello world!
## Application terminated, rc = 0x0
```

其中第一行和最后一行是 `U-boot` 本身输出的，第二行则是我们的例程输出的，输出完毕后将回到 `U-boot` 界面，出现提示符。

说明上述例程的编译链接时，为了展示详细的过程，我们使用了分步命令操作的方法。实际开发中当然会使用 `Makefile`，下面是可用于本例程的一个 `Makefile`：

```
# 文件名: Makefile
# 说明: 固件程序 Makefile

CROSS_COMPILE ?= arm-linux-
GCC := $(CROSS_COMPILE)gcc
LD := $(CROSS_COMPILE)ld
OBJCOPY := $(CROSS_COMPILE)objcopy

TARGET := hello
ASM_SRC := start.S
```



```
C_SRC := hello.c

ASM_OBJS := $(ASM_SRC:.S=.o)
C_OBJS := $(C_SRC:.c=.o)
OBJS := $(ASM_OBJS) $(C_OBJS)

.PHONY: all clean

all: $(TARGET).bin

$(TARGET).bin: $(TARGET)
    $(OBJCOPY) -O binary -S $< $@

$(TARGET): $(OBJS)
    $(LD) -Ttext 0x30000000 $^ libstubs.a -o $@

%.o : %.c
    $(GCC) -c $< -o $@

%.o : %.S
    $(GCC) -c $< -o $@

%.o : %.s
    $(GCC) -c $< -o $@

clean:
    -rm -f $(OBJS)
    -rm -f $(TARGET).bin
    -rm -f $(TARGET)
```

对它稍加修改即可用于其他较复杂的程序。



第 6 章 S3C2410 接口与编程

嵌入式处理器的特点是芯片内除了具有 CPU 核心的计算功能外，还集成了大量的外围控制器件。以 S3C2410A 处理器为例，它是一款为手持设备设计的低功耗、高度集成的嵌入式处理器，采用 ARM920T 内核，并且在芯片内还集成了如下的外围控制器与接口：

- ◆ SDRAM 控制器。
- ◆ LCD 控制器。
- ◆ 3 个通道的 UART (Universal Asynchronous Receiver/Transmitter，通用异步接收与发送器)。
- ◆ 4 个通道的 DMA (Direct Memory Access，直接内存访问)。
- ◆ 4 个具有 PWM (Pulse Width Modulation，脉冲宽度调制) 功能的计时器和 1 个内部时钟。
- ◆ 8 通道的 10 位 ADC/触摸屏接口。
- ◆ I²C 总线及 I²S 接口。
- ◆ 2 个 USB 主机接口或 1 个 USB 主机接口加 1 个 USB 设备接口。
- ◆ 2 个 SPI 接口。
- ◆ SD 卡接口和 MMC 卡接口。
- ◆ 117 位 GPIO (General Purpose I/O，通用输入输出) 端口，其中包括 24 路外部中断源。

因为集成了许多外围的接口，使得嵌入式处理器不需要外扩许多芯片就能形成完整的计算机系统，大大减少了整个系统的体积与复杂度。

在嵌入式开发中，即使是在有操作系统支持的情况下，当开发驱动时，也难免要直接对底层硬件进行控制，因此对处理器芯片内集成的各种控制器有一定了解是必要的。

本章将对 S3C2410A 处理器内比较常用的控制器进行说明，采用先讲述原理、后给出例程的方式。这里的例程都是直接在 U-boot 环境中执行的固件程序。

6.1 软中断异常编程

软中断是处理器提供的一个很有用的异常处理机制，是操作系统的系统调用机制实现的硬件基础。本节将通过实例详细说明软中断异常的整个处理流程。

6.1.1 软中断异常入口

当程序中用 swi 指令触发软中断异常时，处理器将进入管理模式，并且跳转到相应的异常入口地址处执行。当采用低端入口地址时，软中断异常的入口地址是 0x00000008。我们可以参考 U-boot 中异常向量表的设置，其代码如下：

```
.globl _start
_start: b      reset
        ldr    pc, _undefined_instruction
        ldr    pc, _software_interrupt
        ldr    pc, _prefetch_abort
        ldr    pc, _data_abort
        ldr    pc, _not_used
        ldr    pc, _irq
        ldr    pc, _fiq

_undefined_instruction: .word undefined_instruction
_software_interrupt:   .word software_interrupt
_prefetch_abort:      .word prefetch_abort
_data_abort:          .word data_abort
_not_used:            .word not_used
_irq:                 .word irq
_fiq:                 .word fiq
```

从中可以看出，在异常入口地址处放置的是一条跳转指令，跳转的目标则存放在向量表后面的数据中，这些数据的值是真正的处理程序入口地址。这样做有一个好处，我们可以直接修改异常向量表后面的数据区，以达到修改处理程序入口的目的。

swi 指令的操作数对处理器来说没什么意义，但在软件上可以利用它来传递一个参数。Linux 上传统的系统调用接口中就是这样做的，用它来传递系统调用的功能号。这样，在中断服务程序中就需要提取出这个功能号，代码如下：

```
ldr    r0, [lr, #-4]
bic    r0, r0, #0xff000000
```

因为软中断异常发生时，**lr** 寄存器的值被设为 **swi** 指令的下一条指令的地址，因此减去 4 就是 **swi** 指令本身的地址，这样就可以读到 **swi** 指令的内容。它的低 24 位是表示功能号的，因此将其高 8 位清零即可。然后可以根据这个功能号调用不同的函数来进行处理。

软中断异常处理完毕之后，可通过以下指令进行返回：

```
movs   pc, lr
```

注意这里必须用 **movs** 而不能用 **mov**，因为异常返回时要将 **SPSR** 寄存器的内容复制到 **CPSR**，以使处理器回到原来的模式。

6.1.2 软中断异常应用例程

下面我们编写一个固件程序来说明软中断异常的处理流程。因为是固件程序，所以实际上不存在像系统调用那样从用户模式到管理模式的切换，但处理的流程是一致的。

例程分为两个部分，一部分是软中断的处理程序，另一部分则用来对软中断进行测试。

6.1.2.1 软中断处理程序

软中断处理程序由两部分组成，一部分为汇编源代码，另一部分为 C 语言源代码。因为异常

模式的入口与返回不可能用 C 语言描述，所以必须用汇编实现。由这段汇编再去调用 C 语言的函数以实现主要的功能。

汇编源程序的内容如下：

```
@ 文件名: swihandler.S
@ 说明: 软中断异常应用例程

.global _start
.text
.extern c_swi_handler
_start:
    stmfd    sp!, {r0 - r12, lr} @ 寄存器入栈
    mrs      r1, spsr
    stmfd    sp!, {r1} @ spsr 入栈
    ldr      r0, [lr, #-4] @ 得到软中断指令
    bic      r0, r0, #0xff000000 @ 从指令中提取软中断功能号
    bl       c_swi_handler @ 调用 C 函数
    ldmfd    sp!, {r1}
    msr      spsr_cxsf, r1 @ spsr 出栈
    str      r0, [sp] @ 准备返回值, r0 在栈顶
    ldmfd    sp!, {r0 - r12, pc}^ @ 寄存器出栈并返回
.end
```

在上述代码中，首先进行寄存器的保护，然后取出功能号，并传递给 `c_swi_handler` 函数进行处理。在异常的处理程序中一般要对用到的寄存器进行保护。因为异常可以发生在任何时候，如果异常返回时某些寄存器的值发生了改变，则程序执行的上下文将是不可靠的。对于软中断异常来说，有些寄存器用于传递参数和返回值，可以不做保护。保护的方式是先压入栈中，返回时再从栈中弹出。

`c_swi_handler` 函数实现在 C 语言源码中，其内容如下：

```
/* 文件名: c_handler.c */
/* 说明: 软中断异常应用例程 */

int c_swi_handler(unsigned int number)
{
    printf("SWI %d triggered!\n", number);
    switch (number) {
        case 0x10:
            printf("You got it!\n");
            return 0x888;
            break;
        default:
            printf("Not handled!\n");
            return 0;
    }
}
```

我们的处理非常简单，如果功能号是 `0x10`，就输出特殊的信息，并返回一个特殊的值以便于

观察结果。

6.1.2.2 软中断测试程序

为了验证上述中断处理程序，必须有另外一个程序用来发起软中断异常。程序的源码如下：

```
@ 文件名: test_swi.S
@ 说明: 软中断应用例程, 测试

.global _start
.text
_start:
    stmfd    sp!, {lr} @ 保存返回地址
    mov      r1, #0x24 @ 软中断入口地址的位置
    ldr      r2, [r1] @ 读取原来的地址
    stmfd    sp!, {r2} @ 保存原来的地址
    ldr      r2, =0x30002000 @ 新的中断入口地址
    str      r2, [r1] @ 设置新的入口地址
    nop
    swi      0x10
    nop
    ldmdfd   sp!, {r2}
    mov      r1, #0x24
    str      r2, [r1] @ 恢复原来的入口地址
    ldmdfd   sp!, {pc} @ 返回
.end
```

由于我们的程序是在 U-boot 环境中运行的，异常向量表已经事先设置好了，指向 U-boot 自己提供的处理程序，所以必须先将其修改为指向我们提供的处理程序。这样，当软中断异常发生时，执行流程才能进入我们的处理程序。在测试程序退出前，再恢复原来的入口地址。

6.1.2.3 测试过程

首先进行编译，软中断的处理程序编译过程如下：

```
arm-linux-as swihandler.S -o swihandler.o
arm-linux-gcc -c c_handler.c
arm-linux-ld -Ttext 0x30002000 swihandler.o c_handler.o libstubs.a -o handler
arm-linux-objcopy -O binary -S handler handler.bin
```

这样就生成了二进制可执行文件 handler.bin。因为我们在测试程序中将入口地址设为了 0x30002000，故链接地址也设为了 0x30002000。

测试程序的编译过程如下：

```
arm-linux-as test_swi.S -o test_swi.o
arm-linux-ld -Ttext 0x30008000 test_swi.o -o test
arm-linux-objcopy -O binary -S test test.bin
```

这样就生成了二进制可执行文件 test.bin。

然后将文件 handler.bin 下载到目标机内存 0x30002000 地址处，将 test.bin 下载到目标机

内存 0x30008000 处, 用 go 命令执行 test.bin, 则终端上会显示如下结果:

```
## Starting application at 0x30008000 ...
SWI 16 triggered!
You got it!
## Application terminated, rc = 0x888
```

输出的信息表明我们的中断处理程序得到了调用并且正确地提取到了功能号。

6.2 中断控制器及外部中断编程

区别于上一节所说的软中断异常, 这里的中断指的是 ARM 处理器外部的的事件引发的异常。与软中断异常的最大区别是, 程序中不知道何时会发生外部中断异常。

ARM 处理器支持两种类型的外部中断, 即 IRQ 及 FIQ。ARM CPU 核心提供了这样的触发信号接口 (可以理解为两根片内的信号线), 但一般来说要求系统能够处理多个中断源, 并且对中断要进行中断状态以及优先级管理, 这样就需要一个称为中断控制器的硬件。

本节将说明 S3C2410A 处理器的中断体系并通过例程来说明如何对中断控制器进行编程。

6.2.1 中断体系结构

如图 6.1 所示是 S3C2410A 处理器的中断体系结构。整个体系中包含如下几个模块: 中断控制器、ARM920T 核心、内部中断源及外部中断源。处理器的中断控制器, 一般为各个芯片厂家自定义的中断控制逻辑, 与 ARM 核心通过中断信号接口 IRQ 及 FIQ 连接在一起。

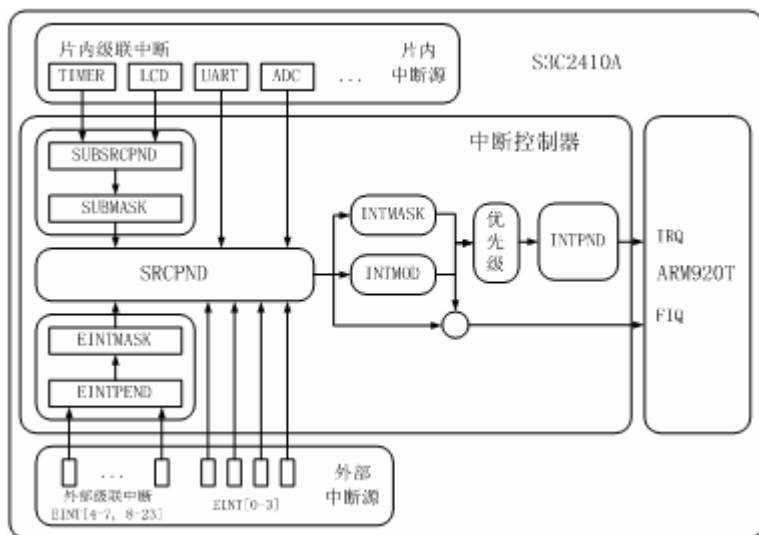


图 6.1 S3C2410A 中断体系

系统内主要有两类中断源: 片内中断源及来自处理器外部的中断输入管脚, 这些管脚接在需要中断支持的外部设备上。

ARM 核本身对中断有控制, 即 CPSR 寄存器的 I 控制位和 F 控制位, 分别表示 IRQ 和 FIQ 的禁用状态。如果这些位被置为 1, 则相应的中断类型将不能发生。因此, 如果要使用中断, 必须

先将 CPSR 程序状态寄存器的 F 位和 I 位清零,当然中断屏蔽寄存器 INTMSK 中相应的位也要清零。

6.2.2 中断控制器

中断控制器用来管理各类不同的中断。各类中断有可能同时发生,但在一个时刻,处理器只能处理一个中断,因此需要中断控制器进行中断源的管理。

当中断控制器检测到有中断发生时,首先将发生中断的中断源记录在中断源标志寄存器 SRCPND 中。软件上可以通过中断屏蔽寄存器 INTMASK 来设置某个中断是否被屏蔽。如果发生的中断被屏蔽,则中断控制器不会向 ARM 核心发起中断请求。

对于同时发生的中断,中断控制器通过其优先级选择模块,按照预先配置的优先级策略来选择高优先级的中断。中断的优先级可以通过配置寄存器 PRIORITY 来改变。选择好需要处理的中断后,中断控制器将中断标志寄存器 INTPND 中的对应位设置为 1,然后向 ARM 核心发起中断请求。在中断处理程序中,可以通过读取 INTPND 寄存器的值来判断是哪个中断需要响应。要注意 SRCPND 和 INTPND 寄存器中的相应位一般不会自动清零,因此需要在中断处理程序中,先清零 SRCPND 的相应位,再清零 INTPND 的相应位,否则 ARM 核心会被反复中断。

通过配置中断模式寄存器 INTMOD 可以控制某个中断是 IRQ 还是 FIQ,但是系统中同时只能有一个中断被配置为 FIQ。

此外,S3C2410A 的中断控制器提供了一个寄存器 INTOFFSET 以方便中断处理程序提取发生中断的中断源,它的值就是 SRCPND 中相关的中断位的位置。例如,定时器 0 的中断 INT_TIMER0 在 SRCPND 中安排在第 10 个比特位,因此当定时器 0 发生中断时,INTOFFSET 寄存器的值就是 10。

以上所述主要是针对 S3C2410A 的一级中断的。实际上 S3C2410 还有许多级联中断,这些级联中断都有相应的级联中断控制逻辑。

对于片内级联中断,配置类似于一级中断,有相关的屏蔽设置。但是,中断处理程序中得到中断源的过程会复杂一些,仅仅依靠 INTPND 或 INTOFFSET 寄存器是不能准确判定具体哪一个中断源需要响应的,必须通过级联控制逻辑中的 SUBSRCPND 寄存器才能进一步确定。

对于片外的级联中断,其配置更复杂一些。首先很多片外中断的管脚是与 GPIO 复用的,必须先配置相应的管脚为中断方式,然后还要配置其触发的信号特性,比如是电平触发还是边缘触发。对于不同的触发类型还必须配置其各自的属性:边沿触发需要配置是上升沿触发还是下降沿触发,电平触发还需配置其可靠的电平触发窗口时间。在中断处理程序中,与片内级联中断一样,不能仅仅根据 INTPND 及 INTOFFSET 寄存器判断具体哪个管脚发生了中断,必须根据相关的寄存器进一步进行判断。

6.2.3 中断源安排

S3C2410A 共有 56 个中断源,其中 30 个为一级中断源,使用一个 32 位寄存器即可进行管理控制。在这 30 个一级中断源里,有片内片外的级联中断,如外部中断 EINT4 - EINT7 这 4 个中断源通过“或”的方式提供一个级联中断源送至控制器的 SRCPND。同样,EINT8 - EINT23 这 16 个中断源也是如此。S3C2410A 中断源的详细安排如表 6.1 所示。

表 6.1 S3C2410A 中断源安排

中断源	描述	比特位/中断号
INT_ADC	A/D 转换结束中断	31
INT_RTC	实时时钟中断	30
INT_SPI	同步串行设备 1 中断	29
INT_UART0	异步串口 0 中断	28
INT_IIC	I2C 中断	27
INT_USBH	USB 主机控制器中断	26
INT_USBD	USB 设备中断	25
Reserved	保留	24
INT_UART1	异步串口 1 中断	23
INT_SP0	同步串行设备 0 中断	22
INT_SDI	SDI 中断	21
INT_DMA3	DMA3 通道中断	20
INT_DMA2	DMA2 通道中断	19
INT_DMA1	DMA1 通道中断	18
INT_DMA0	DMA0 通道中断	17
INT_LCD	LCD 中断	16
INT_UART2	异步串口 2 中断	15
INT_TIMER4	定时器 4 中断	14
INT_TIMER3	定时器 3 中断	13
INT_TIMER2	定时器 2 中断	12
INT_TIMER1	定时器 1 中断	11
INT_TIMER0	定时器 0 中断	10
INT_WDT	看门狗定时器中断	9
INT_TICK	实时时钟时间滴答中断	8
NBATT_FLT	电池故障中断	7
Reserved	保留	6
EINT8-EINT23	外部中断 8 - 23	5
EINT4-EINT7	外部中断 4 - 7	4
EINT3	外部中断 3	3
EINT2	外部中断 2	2
EINT1	外部中断 1	1
EINT0	外部中断 0	0

我们可以使用各个中断源被安排的比特位作为它们的中断号使用。例如，第 10 位安排的是 TIMER0 定时器中断，于是 10 可以作为它的中断号。实际上，INTOFFSET 寄存器中记录的数值即为 10，也可以根据它来确定是定时器 0 发生了中断。



6.2.4 中断控制器寄存器配置

S3C2410A 中断控制器主要的寄存器被映射在从 0x4A000000 开始的内存空间上,有以下 8 个。

- ◆ 中断源标志寄存器 SRCPND: 它的每个比特位用来记录对应的中断是否已经发生。
- ◆ 中断模式寄存器 INTMOD: 记录每个中断的模式。
- ◆ 中断屏蔽寄存器 INTMSK: 它的每个比特位用来表示对应的中断是否被屏蔽。
- ◆ 中断优先级控制寄存器 PRIORITY: 用于控制中断的优先级。
- ◆ 中断标志寄存器 INTPND: 用来表示通过优先级选择后应该被响应的中断。
- ◆ 片内次级中断源标志寄存器 SUBSRCPND: 记录次级中断源的请求状态。
- ◆ 片内次级中断屏蔽寄存器 INTSUBMSK: 记录次级中断源屏蔽状态。
- ◆ 中断位置寄存器 INTOFFSET: 记录所发起的中断的位置。

而与次级的外部中断有关的寄存器被映射在从 0x56000000 开始的内存空间上,主要有如下几个。

- ◆ 外部中断控制寄存器 EXTINT0, EXTINT1 及 EXTINT2: 每三位表示一个中断的触发方式。
- ◆ 外部中断屏蔽寄存器 EINTMASK: 它的每个比特位表示对应的中断是否被屏蔽。
- ◆ 外部中断标志寄存器 EINTPEND: 它的每个比特位表示对应的中断源发起了请求。

各寄存器的具体说明如表 6.2 所示。

表 6.2 S3C2410A 中断控制器寄存器

寄存器	地址	读 / 写	描述	初始值
SRCPND	0x4A000000	R/W	0 = 没有中断请求, 1 = 中断源发出中断请求	0x00000000
INTMOD	0x4A000004	R/W	0 = IRQ 模式, 1 = FIQ 模式	0x00000000
INTMSK	0x4A000008	R/W	0 = 允许响应中断请求, 1 = 屏蔽中断请求	0xFFFFFFFF
PRIORITY	0x4A00000C	R/W	优先级控制	0x7F
INTPND	0x4A000010	R/W	0 = 不需要响应的中断, 1 = 需要响应的中断	0x00000000
INTOFFSET	0x4A000014	R/W	指示中断请求的中断号	0x00000000
SUBSRCPND	0x4A000018	R/W	0 = 发生中断请求, 1 = 没有中断请求发生	0x00000000
INTSUBMSK	0x4A00001C	R/W	0 = 允许响应中断源, 1 = 中断请求被屏蔽	0x7FF
EXTINT0	0x56000088	R/W	000 = 低电平触发, 001 = 高电平触发, 01x = 下降沿触发, 10x = 上升沿触发, 11x = 边沿触发	0x00000000
EXTINT1	0x5600008C	R/W	同上	0x00000000
EXTINT2	0x56000090	R/W	同上	0x00000000
EINTMASK	0x560000A4	R/W	0 = 允许外部中断, 1 = 屏蔽外部中断	0x0FFFFFF0
EINTPEND	0x560000A8	R/W	0 = 外部中断发出请求, 1 = 无外部中断请求	0x00000000

S3C2410A 共有 24 个次级外部中断,它们可以有多种触发方式,由 EXTINT0, EXTINT1 和



EXTINT2 三个寄存器进行设置，这些寄存器每 4 个比特位控制一个中断源，但其中只有低 3 位是有效的。屏蔽标志则通过 EINTMASK 寄存器设置。

6.2.5 中断应用例程

下面我们以一个例程来说明中断应用开发中对硬件的配置过程。我们使用 HY2410A 开发板的键盘来触发中断，它使用外部中断 EINT3，对应着 S3C2410A 的 GPF3 管脚。当键按下时，管脚上的电平将由高变低；当键松开时，管脚上的电平将由低变高。这样，将中断触发类型设置为下降沿触发即可检测到键按下的事件。

这个中断应用与上一节的软中断应用类似，也划分为两个部分：一部分是中断处理程序，另一部分是中断测试程序。

6.2.5.1 中断测试程序

首先我们将中断相关的寄存器定义在一个头文件中，以便在其他 C 源程序中使用。头文件的源码如下：

```
/* 文件名: regs_irq.h */
/* 说明: 外部中断例程寄存器定义 */

#ifndef REGS_IRQ_INCLUDED
#define REGS_IRQ_INCLUDED

#define rGPFCON      (*(volatile unsigned int*)0x56000050)
#define rGPFDAT      (*(volatile unsigned int*)0x56000054)
#define rGPFUP       (*(volatile unsigned int*)0x56000058)

#define rGPGCON      (*(volatile unsigned int*)0x56000060)
#define rGPGDAT      (*(volatile unsigned int*)0x56000064)
#define rGPGUP       (*(volatile unsigned int*)0x56000068)

#define rEXTINT0      (*(volatile unsigned int*)0x56000088)
#define rEXTINT1      (*(volatile unsigned int*)0x5600008c)
#define rEXTINT2      (*(volatile unsigned int*)0x56000090)
#define rEINTMASK     (*(volatile unsigned int*)0x560000a4)
#define rEINTPND      (*(volatile unsigned int*)0x560000a8)

#define rSRCPND       (*(volatile unsigned int*)0x4a000000)
#define rINTMOD       (*(volatile unsigned int*)0x4a000004)
#define rINTMSK       (*(volatile unsigned int*)0x4a000008)
#define rPRIORITY     (*(volatile unsigned int*)0x4a00000c)
#define rINTPND       (*(volatile unsigned int*)0x4a000010)
#define rINTOFFSET    (*(volatile unsigned int*)0x4a000014)
#define rSUBSRCPND    (*(volatile unsigned int*)0x4a000018)
#define rINTSUBMSK    (*(volatile unsigned int*)0x4a00001c)

#endif
```

按照这样的定义，在 C 源程序中可以直接把这些宏作为全局变量来进行读写。

中断测试程序要对中断控制器进行配置操作，设置好中断控制器相关的寄存器并使能中断，这些操作在一个 C 源程序内完成。另外它还包括一段汇编代码，用于替换原来的异常入口地址。其汇编源文件的内容如下：

```
@ 文件名: testirq.S
@ 说明: 中断应用例程, 测试

.global _start
.text
.extern irq_init
.extern irq_exit
_start:
    stmfd    sp!, {lr} @ 保存返回地址
    mov     r1, #0x34 @ 中断入口地址的位置
    ldr     r2, [r1] @ 读取原来的地址
    stmfd    sp!, {r2} @ 保存原来的地址
    ldr     r2, =0x30002000 @ 新的中断入口地址
    str     r2, [r1] @ 设置新的入口地址
    @ 将地址 0x30000000 处的值设为 1
    ldr     r1, =0x30000000
    mov     r2, #1
    str     r2, [r1]
    @ 初始化中断
    bl      irq_init
    @ 反复检测地址 0x30000000 处是否已变为 0
    ldr     r1, =0x30000000
1:
    ldr     r2, [r1]
    teq     r2, #0
    bne     1b
    @ 退出
    bl      irq_exit
    ldmfd    sp!, {r2}
    mov     r1, #0x34
    str     r2, [r1] @ 恢复原来的入口地址
    ldmfd    sp!, {pc} @ 返回
.end
```

在这段代码中，当所有的初始化操作完成后，程序将把地址 0x30000000 处的值设为 1，然后进入一个循环，循环中反复检测这个地址处的值是否已变为 0，如果为 0 则退出循环。显然，如果没有其他手段修改地址 0x30000000 处的值，则程序将陷入死循环。我们把修改的操作放在中断处理程序中，这样当中断发生时，测试程序才会退出。

中断初始化的操作放在 C 源程序中，其内容如下：

```
/* 文件名: c_testirq.c */
/* 说明: 中断应用例程, 测试 */

#include "regs_irq.h"
```



```

/* 使能中断 */
static void enable_interrupts(void)
{
    unsigned long temp;
    __asm__ __volatile__(
        "mrs %0, cpsr\n\t"
        "bic %0, %0, #0x80\n\t"
        "msr cpsr_c, %0\n\t"
        : "=r" (temp)
        :
        : "memory");
}

/* 禁用中断 */
static void disable_interrupts(void)
{
    unsigned long temp;
    __asm__ __volatile__(
        "mrs %0, cpsr\n\t"
        "orr %0, %0, #0x80\n\t"
        "msr cpsr_c, %0\n\t"
        : "=r" (temp)
        :
        : "memory");
}

/* 中断初始化 */
void irq_init()
{
    /* 配置中断寄存器 */
    rINTMOD = 0x00; /* 设置所有中断为 IRQ 模式 */
    rINTMSK = 0xffffffff; /* 屏蔽所有中断 */
    rGPFUP = 0xFFFF; /* 禁止 GPF 组的 GPIO 管脚上拉 */
    /* 配置 GPF3 管脚为中断方式 */
    rGPFCON = rGPFCON | (1 << 7) & ~(1 << 6); /* GPFCON[7:6] = 10 */
    /* 设置 GPF3 为下降沿触发 */
    rEXTINT0 = rEXTINT0 & ~0xF000u | 0x2000u; /* EXTINT0[14:12] = 010 */
    /* 清除所有中断标志 */
    rSRCPND = 0xFFFFFFFF;
    rINTPND = 0xFFFFFFFF;
    /* 清除 EINT3 屏蔽位 */
    rINTMSK &= ~(1 << 3);
    /* 使能中断 */
    enable_interrupts();
}

void irq_exit()
{
    disable_interrupts();
}

```

6.2.5.2 中断处理程序

中断处理程序也包含两个部分，一个是汇编程序 `irqhandler.S`，另一个是 C 源程序 `c_irqhandler.c`。

`irqhandler.S` 文件的源码如下：

```
@ 文件名: irqhandler.S
@ 说明: 中断应用例程, 中断处理程序入口

.global _start
.extern do_irq
.text
_start:
    ldr        sp, =0x3000A000 @ 设置中断栈
    stmfd     sp!, {r0 - r12, lr} @ 寄存器入栈
    mrs       r1, spsr
    stmfd     sp!, {r1} @ spsr 入栈
    bl        do_irq
    ldmfd     sp!, {r1}
    msr       spsr_cxsf, r1 @ spsr 出栈
    ldmfd     sp, {r0 - r12, lr} @ 寄存器出栈
    subs     pc, lr, #4 @ 返回
.end
```

因为 U-boot 运行时不需要处理中断，所以一般没有为中断模式设置栈，故我们需要自己初始化中断栈，即将栈指针指向内存中一块没有使用的区域。与软中断不同，中断返回时需要将 PC 寄存器的值设为 LR 寄存器的值减 4。

`c_irqhandler.c` 文件的内容如下：

```
/* 文件名: c_irqhandler.c */
/* 说明: 外部中断应用例程, 中断处理程序 */

#include "regs_irq.h"

void do_irq(void)
{
    unsigned int irqno = rINTOFFSET; /* 得到中断号 */
    printf("The IRQ number is %d\n", irqno);
    /* 清除所有中断标志 */
    rSRCPND = 0xFFFFFFFF;
    rINTPND = 0xFFFFFFFF;
    /* 将内存地址 0x30000000 处写为 0 */
    (*(volatile unsigned long*)0x30000000) = 0;
}
```

这段代码中只是输出了一个中断号，表明中断确实发生了，并且执行到了我们提供的中断处理程序。特别要注意清除各种中断标志，否则中断将反复发生。

6.2.5.3 测试过程

首先编译中断处理程序，所用的命令如下：

```
arm-linux-gcc -c irqhandler.S -o irqhandler.o
arm-linux-gcc -c c_irqhandler.c -o c_irqhandler.o
arm-linux-ld -Ttext 0x30002000 irqhandler.o c_irqhandler.o libstubs.a -o
    handler
arm-linux-objcopy -O binary -S handler handler.bin
```

生成二进制可执行程序 **handler.bin**。然后编译中断测试程序，所用命令如下：

```
arm-linux-gcc -c testirq.S -o testirq.o
arm-linux-gcc -c c_testirq.c -o c_testirq.o
arm-linux-ld -Ttext 0x30008000 testirq.o c_testirq.o -o test
arm-linux-objcopy -O binary -S test test.bin
```

生成二进制可执行程序 **test.bin**。

将 **handler.bin** 下载至目标机内存 **0x30002000** 处，将 **test.bin** 下载至 **0x30008000** 处，用 **go** 命令执行 **test.bin**，将显示如下结果：

```
## Starting application at 0x30008000 ...
```

这时 **U-boot** 界面将失去响应，因为处理器已陷入死循环，按下键盘上的某个键，则继续显示如下结果：

```
The IRQ number is 3
## Application terminated, rc = 0x1
```

输出的信息说明提取到的中断号是 **3**，并且测试程序正常退出了。

6.3 定时器及其编程

S3C2410A 处理器共有 5 个 16 位的定时器。其中定时器 0, 1, 2, 3 具有 PWM 功能，它们各有一个输出引脚，能够用于产生占空比可调的脉冲宽度调制信号。定时器 4 没有对应的输出管脚，只能用于内部的定时。

本节将介绍 **S3C2410A** 的定时器体系结构及硬件配置方法，并通过实例来具体说明如何对定时器进行编程。

6.3.1 定时器体系

S3C2410A 的定时器体系结构如图 6.2 所示。

定时器的时钟源是 **PCLK**，在 **HY2410A** 开发板上，它的频率是 50MHz，由 200MHz 的 **FCLK** 进行 4 分频而得到。**PCLK** 时钟源首先通过可编程的 8 位预分频器做第一次分频，降低频率，然后再经过一次固定比例的时钟分频器，这个分频器同时提供 5 种输出：1/2, 1/4, 1/8, 1/16 及外部时钟 **TCLK**。每个定时器模块从时钟分频器接收它自己的时钟信号。

5 个定时器分为两路，定时器 **T0** 和 **T1** 共用一套 8 位预分频器和时钟分频器，定时器 **T2**, **T3**, **T4** 共用另一套 8 位预分频器和时钟分频器。

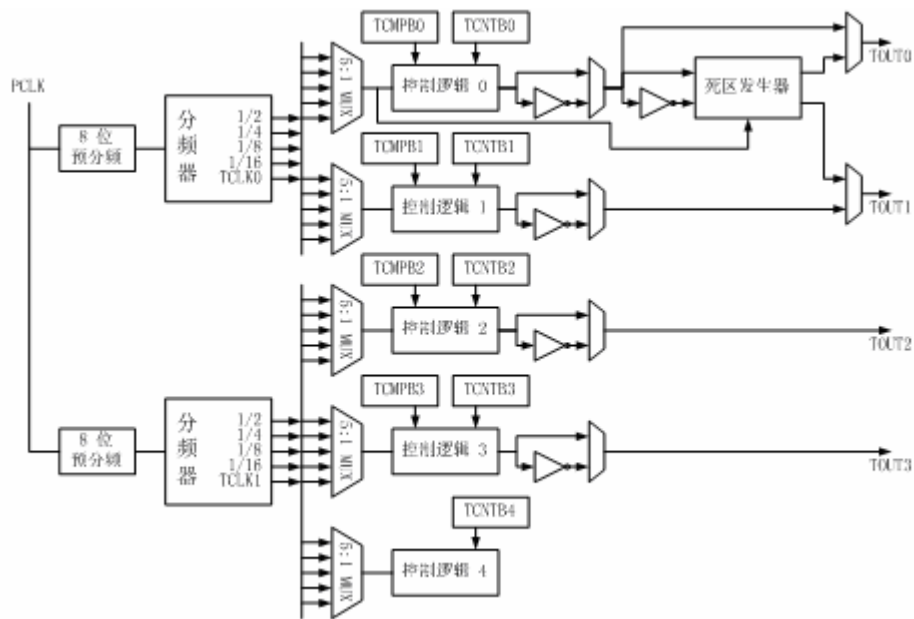


图 6.2 S3C2410A 定时器体系结构

6.3.2 定时器单元工作原理

定时器单元使用分频得到的输入时钟驱动其 16 位的减法计数器，每过一个时钟周期则计数

器值减 1，当计数器值减到 0 时就触发一个定时器中断，定时操作完成。

通过配置定时器控制寄存器 TCON 可以决定当定时完成时是否自动重载定时器值。如果配置了自动重载，那么当定时操作完成时，相应的计数寄存器 TCNTB_n 的值被自动重新载入到减法计数器中，并继续下次定时操作。通过清除定时器控制寄存器 TCON 中的定时器使能位可以终止定时器的运行。

定时器比较寄存器 TCMPB_n 的值用于产生脉冲宽度可调的输出信号，当这个值与定时器的减法计数器值相匹配时，定时器的输出电平将改变。因此，比较寄存器的值决定了 PWM 信号脉冲的占空比，这个值可以通过软件来修改。

综上所述可以看出，定时器的一个定时周期 T 与下述两个值相关：

- ◆ 定时器的计数器初始值，由 TCNTB_n 的值决定
- ◆ 定时单元的输入时钟频率 F_{tclk}

它们之间的关系如下：

$$T = F_{tclk} / TCNTB_n$$

而定时单元的输入时钟频率 F_{tclk} 又与以下三个值相关：

- ◆ PCLK 时钟的频率 F_{pclk}
- ◆ 预分频器的分频值 T_{pre}

它们之间的关系如下：

$$F_{\text{tclk}} = F_{\text{pclk}} / ((T_{\text{pre}} + 1) \times T_{\text{div}})$$

当然，当时钟分频器选择为 $\text{TCLK}n$ 时，它的频率由相应的外部时钟的频率决定。

6.3.3 定时器寄存器配置

与定时器有关的寄存器被映射在从地址 $0x51000000$ 开始的内存空间上，如表 6.3 所示。

表 6.3 定时器相关的寄存器

寄存器	地址	读/写	描述	初始值
TCFG0	0x51000000	R/W	定时器配置寄存器 0	0x00000000
TCFG1	0x51000004	R/W	定时器配置寄存器 1	0x00000000
TCON	0x51000008	R/W	定时器控制寄存器	0x00000000
TCNTB0	0x5100000C	R/W	定时器 0 计数缓冲寄存器	0x00000000
TCMPB0	0x51000010	R/W	定时器 0 比较缓冲寄存器	0x00000000
TCNTO0	0x51000014	R	定时器 0 计数观察寄存器	0x00000000
TCNTB1	0x51000018	R/W	定时器 1 计数缓冲寄存器	0x00000000
TCMPB1	0x5100001C	R/W	定时器 1 比较缓冲寄存器	0x00000000
TCNTO1	0x51000020	R	定时器 1 计数观察寄存器	0x00000000
TCNTB2	0x51000024	R/W	定时器 2 计数缓冲寄存器	0x00000000
TCMPB2	0x51000028	R/W	定时器 2 比较缓冲寄存器	0x00000000
TCNTO2	0x5100002C	R	定时器 2 计数观察寄存器	0x00000000
TCNTB3	0x51000030	R/W	定时器 3 计数缓冲寄存器	0x00000000
TCMPB3	0x51000034	R/W	定时器 3 比较缓冲寄存器	0x00000000
TCNTO3	0x51000038	R	定时器 3 计数观察寄存器	0x00000000
TCNTB4	0x5100003C	R/W	定时器 4 计数缓冲寄存器	0x00000000
TCNTO4	0x51000040	R	定时器 4 计数观察寄存器	0x00000000

注意 S3C2410A 的定时器计数器采用双缓冲技术，即一个寄存器用于计数，而另一个寄存器用于设置计数值，所设置的计数值在下次开始计数时才装入用于计数的寄存器，因此对计数值的设置不影响正在进行的计数过程。对应于每个定时器有一个寄存器 $\text{TCNTB}n$ 是用来设置计数值的，而通过另一个寄存器 $\text{TCNTO}n$ 可得到当前的计数值，它是一个只读的寄存器。除了定时器 4 外，其他定时器还有一个 $\text{TCMPB}n$ 寄存器用于控制输出的占空比。

定时器配置寄存器 0 (TCFG0) 各个比特位的具体含义如表 6.4 所示。

表 6.4 定时器配置寄存器 0 说明

比特位	含义	描述
31 - 24	保留	无
23 - 16	死区长度	控制死区长度
15 - 8	预分频器 1	定时器 2, 3, 4 的预分频值
7 - 0	预分频器 0	定时器 0, 1 的预分频值



定时器配置寄存器 1（TCFG1）各个比特位的具体含义如表 6.5 所示。

表 6.5 定时器配置寄存器 1 说明

比特位	含义	描述
31 - 24	保留	无
23 - 20	DMA 模式选择	0000 = 未选择, 0001 = 选择定时器 0, 0010 = 选择定时器 1, 0011 = 选择定时器 2, 0100 = 选择定时器 3, 0101 = 选择定时器 4, 0110 = 保留
19 - 16	定时器 4 时钟选择	0000 = 1/2, 0001 = 1/4, 0010 = 1/8, 0011 = 1/16, 01xx = TCLK1
15 - 12	定时器 3 时钟选择	同上
11 - 8	定时器 2 时钟选择	同上
7 - 4	定时器 1 时钟选择	0000 = 1/2, 0001 = 1/4, 0010 = 1/8, 0011 = 1/16, 01xx = TCLK0
3 - 0	定时器 0 时钟选择	同上

定时器控制寄存器（TCON）各个比特位的具体含义如表 6.6 所示。

表 6.6 定时器控制寄存器说明

比特位	含义	描述
22	定时器 4 自动重载标志	0 = 只运行 1 次, 1 = 自动重载
21	定时器 4 手动更新	0 = 无操作, 1 = 更新计数器值
20	定时器 4 启动标志	0 = 停止, 1 = 启动
19	定时器 3 自动重载标志	0 = 只运行 1 次, 1 = 自动重载
18	定时器 3 输出倒相	0 = 不倒相, 1 = 输出倒相
17	定时器 3 手动更新	0 = 无操作, 1 = 更新计数器值
16	定时器 3 启动标志	0 = 停止, 1 = 启动
15	定时器 2 自动重载标志	0 = 只运行 1 次, 1 = 自动重载
14	定时器 2 输出倒相	0 = 倒相关闭, 1 = 输出倒相
13	定时器 2 手动更新	0 = 无操作, 1 = 更新计数器值
12	定时器 2 启动标志	0 = 停止, 1 = 启动
11	定时器 1 自动重载标志	0 = 只运行 1 次, 1 = 自动重载
10	定时器 1 输出倒相	0 = 倒相关闭, 1 = 输出倒相
9	定时器 1 手动更新	0 = 无操作, 1 = 更新计数器值
8	定时器 1 启动标志	0 = 停止, 1 = 启动
7 - 5	保留	无
4	死区功能允许标志	0 = 禁止, 1 = 允许
3	定时器 0 自动重载标志	0 = 只运行 1 次, 1 = 自动重载
2	定时器 0 输出倒相	0 = 倒相关闭, 1 = 输出倒相
1	定时器 0 手动更新	0 = 无操作, 1 = 更新计数器值
0	定时器 0 启动标志	0 = 停止, 1 = 启动



其中手动更新标志位用于将 $TCNTB_n$ 和 $TCMPB_n$ 寄存器的值装入定时器的内部控制逻辑。这个标志位必须由 0 变成 1 时才起作用。

定时器的输出管脚 $TOUT_n$ 在初始状态时为高电平，若无倒相设置，则启动定时器工作时， $TOUT_n$ 管脚输出低电平，然后在比较点翻转为高电平，最后当计数减到 0 时，触发定时器中断， $TOUT_n$ 也同时恢复到低电平。如果设置了倒相，则管脚的电平状态与未设置时完全相反。因此，控制定时器的倒相标志位，就可以控制定时器开始工作时的输出电平，通过调节 $TCNTB_n$ 的值可以控制输出波形的占空比。

一般来说，操作定时器可以分为如下几个步骤。

- step 1** 设置预分频。
- step 2** 设置时钟分频。
- step 3** 设置定时器的 $TCNTB_n$ 和 $TCMPB_n$ 寄存器。
- step 4** 配置定时器控制寄存器。
- step 5** 启动定时器。

6.3.4 定时器应用例程

由于定时器中断仍然是硬件中断，受中断控制器的管理，因此我们可以在上一节中断应用例程的基础上开发定时器的应用。

在 HY2410A 开发板上的 D1, D2, D3, D4 四个 LED 发光二极管都接在 S3C2410A 处理器的 GPIO 管脚上，具体的对应关系如下。

- ◆ GPG0 对应 D1。
- ◆ GPF3 对应 D2。
- ◆ GPF4 对应 D3。
- ◆ GPF7 对应 D4。

由于这些 LED 的另一端都接在电源电压上，如果配置 GPIO 口输出为高，LED 不点亮；反之，当 GPIO 口输出为低时，对应的 LED 点亮。我们可以设置定时器为自动重载方式，并在定时器中断的处理程序中反复改变某个 GPIO 口的输出电平，达到让对应的 LED 闪烁的目的。注意，GPF3 同时用做键盘中断，因此我们选择控制 GPF7。由于不需要输出脉宽调制信号，可以使用定时器 4。

例程仍然分为两个部分：定时器中断处理程序和定时器测试程序。

6.3.4.1 定时器测试程序

首先，对于定时器相关的寄存器提供一个新的头文件，内容如下：

```
/* 文件名: regs_timer.h */
/* 说明: 定时器寄存器定义 */

#ifndef REGS_TIMER_INCLUDED
#define REGS_TIMER_INCLUDED

#define rTCFG0 (*(volatile unsigned int *)0x51000000)
```



```

#define rTCFG1 (*(volatile unsigned int *)0x51000004)
#define rTCON (*(volatile unsigned int *)0x51000008)
#define rTCNTB0 (*(volatile unsigned int *)0x5100000c)
#define rTCMPB0 (*(volatile unsigned int *)0x51000010)
#define rTCNTO0 (*(volatile unsigned int *)0x51000014)
#define rTCNTB1 (*(volatile unsigned int *)0x51000018)
#define rTCMPB1 (*(volatile unsigned int *)0x5100001c)
#define rTCNTO1 (*(volatile unsigned int *)0x51000020)
#define rTCNTB2 (*(volatile unsigned int *)0x51000024)
#define rTCMPB2 (*(volatile unsigned int *)0x51000028)
#define rTCNTO2 (*(volatile unsigned int *)0x5100002c)
#define rTCNTB3 (*(volatile unsigned int *)0x51000030)
#define rTCMPB3 (*(volatile unsigned int *)0x51000034)
#define rTCNTO3 (*(volatile unsigned int *)0x51000038)
#define rTCNTB4 (*(volatile unsigned int *)0x5100003c)
#define rTCNTO4 (*(volatile unsigned int *)0x51000040)

#define HANDLER_BASE_ADDR ((volatile unsigned int *)0x30001000)

#endif

```

定时器测试程序由汇编源文件 `testirq.S` 和 C 源文件 `c_testtimer.c` 组成，其中 `testirq.S` 文件与上一节的中断测试程序中相同，而 `c_testtimer.c` 的内容如下：

```

/* 文件名: c_testtimer.c */
/* 说明: 定时器应用例程, 测试 */

#include "regs_irq.h"
#include "regs_timer.h"

/* 使能中断 */
static void enable_interrupts(void)
{
    unsigned long temp;
    __asm__ __volatile__(
        "mrs %0, cpsr\n\t"
        "bic %0, %0, #0x80\n\t"
        "msr cpsr_c, %0\n\t"
        : "=r" (temp)
        :
        : "memory");
}

/* 禁用中断 */
static void disable_interrupts(void)
{
    unsigned long temp;
    __asm__ __volatile__(
        "mrs %0, cpsr\n\t"
        "orr %0, %0, #0x80\n\t"
        "msr cpsr_c, %0\n\t"

```




```

        : "=r" (temp)
        :
        : "memory");
    }

/* 装载中断函数地址 */
void request_irq(int irq_no, void (*irq_routine)())
{
    HANDLER_BASE_ADDR[irq_no] = (unsigned int)irq_routine;
}

/* 中断处理 */
void irq_handle_timer()
{
    printf("toggle LED.\n");
    if ((rGPFDAT & 0x80) == 0) /* 如果 GPF7 原来是低电平 */
        rGPFDAT |= 0x80; /* 将其设为高电平 */
    else /* 否则 */
        rGPFDAT &= 0x7f; /* 将其设为低电平 */
}

/* 中断初始化 */
void irq_init()
{
    /* 配置中断寄存器 */
    rINTMOD = 0x00; /* 设置所有中断为 IRQ 模式 */
    rINTMSK = 0xffffffff; /* 屏蔽所有中断 */
    rGPFUP = 0xffff; /* 禁止 GPF 组的 GPIO 管脚上拉 */
    /* 配置 GPF3 管脚为中断方式 */
    rGPFCON = rGPFCON | (1 << 7) & ~(1 << 6); /* GPFCON[7:6] = 10 */
    /* 设置 GPF3 为下降沿触发 */
    rEXTINT0 = rEXTINT0 & ~0xf000u | 0x2000u; /* EXTINT0[14:12] = 010 */
    /* 清除所有中断标志 */
    rSRCPND = 0xffffffff;
    rINTPND = 0xffffffff;
    /* 清除 EINT3 屏蔽位 */
    rINTMSK &= ~(1 << 3);
    /* 配置定时器 */
    rTCFG0 = 0x0000ff00; /* 设置预分频器 1 */
    rTCFG1 = 0x00030000; /* 定时器 4 选择 1/16 时钟分频 */
    rTCNTB4 = 6103; /* 定时器 4 计数器初始值 (1s) */
    rTCON |= (1 << 21); /* 装载计数器值 */
    rTCON = 0x00500000; /* 设置定时器为自动重载并启动 */
    request_irq(14, irq_handle_timer);
    /* 设置 GPF7 管脚为输出方式 */
    rGPFCON = rGPFCON & ~(1 << 15) | (1 << 14);
    /* 清除 INT_TIMER4 屏蔽位 */
    rINTMSK &= ~(1 << 14);
    /* 使能中断 */
    enable_interrupts();
}

```



```
void irq_exit()
{
    disable_interrupts();
    rTCON = 0x00000000; /* 停止定时器 */
}
```

在这里我们采用了一种技巧，将一个函数的入口地址放在固定地址的一个数据表中，在中断处理程序中可以从这个数据表中取出函数地址以便执行。这样，中断服务程序可以比较通用，它的作用就是根据中断号从表中取得函数地址，然后调用这个函数。但是这样一来，我们的测试程序就必须在内存中有固定的位置，不再是位置无关的。

6.3.4.2 定时器中断处理程序

定时器中断处理程序由汇编源文件 `irqhandler.S` 和 C 源文件 `c_irqhandler.c` 组成，其中 `irqhandler.S` 文件与上一节的中断处理程序中相同，而 `c_irqhandler.c` 文件的内容修改如下：

```
/* 文件名: c_irqhandler.c */
/* 说明: 外部中断应用例程，中断处理程序 */

#include "regs_irq.h"
#include "regs_timer.h"

void do_irq(void)
{
    unsigned int irqno = rINTOFFSET; /* 得到中断号 */
    printf("The IRQ number is %d\n", irqno);
    /* 清除中断标志 */
    rSRCPND = (1 << irqno);
    rINTPND = (1 << irqno);
    if (irqno != 3) {
        /* 从中断函数地址表中取出函数地址并执行 */
        (((void (*)(void))(HANDLER_BASE_ADDR[irqno])))(void);
    } else {
        /* 将内存地址 0x30000000 处写为 0 */
        (*(volatile unsigned long*)0x30000000) = 0;
    }
}
```

其中仍然保留了原来的中断处理程序的所有功能，只是增加了从中断函数地址表中取出函数地址并执行的功能。

6.3.4.3 测试过程

首先编译定时器测试程序，所用命令如下：

```
arm-linux-gcc -c testirq.S -o testirq.o
arm-linux-gcc -c c_testtimer.c -o c_testtimer.o
arm-linux-ld -Ttext 0x30008000 testirq.o c_testtimer.o libstubs.a -o test
arm-linux-objcopy -O binary -S test test.bin
```



生成二进制可执行程序 `test.bin`。然后编译定时器中断处理程序，所用命令如下：

```
arm-linux-gcc -c irqhandler.S -o irqhandler.o
arm-linux-gcc -c c_irqhandler.c -o c_irqhandler.o
arm-linux-ld -Ttext 0x30002000 irqhandler.o c_irqhandler.o libstubs.a -o
    handler
arm-linux-objcopy -O binary -S handler handler.bin
```

生成二进制可执行程序 `handler.bin`。

将 `handler.bin` 下载至目标机内存 `0x30002000` 处，`test.bin` 下载至 `0x30008000` 处，用 `go` 命令执行 `test.bin`，将显示如下结果：

```
## Starting application at 0x30008000 ...
The IRQ number is 14
toggle LED.
The IRQ number is 14
toggle LED.
```

程序执行过程中能够看到发光二极管 `D4` 在闪烁。

6.4 GPIO 接口

GPIO 的含义是通用 IO 接口，它可以方便地用于扩展接口，连接外部设备或控制电路，体现了处理器的扩展能力。现代的嵌入式处理器，同时集成了众多的控制器接口，要在有限面积上封装这么多的管脚，并不是一件容易的事情，所以处理器的管脚是比较紧张的资源，往往需要对这些管脚进行复用。

S3C2410A 处理器拥有 117 位可编程的 GPIO 端口，这些端口是复合功能的，即一个管脚可以用于输入、输出或其他特殊功能，具体是什么功能可由软件来配置。这些 GPIO 端口被分为 8 个组，列举如下。

- ◆ GPA: 23 个输出端口及特殊功能。
- ◆ GPB: 11 个输入/输出端口及特殊功能。
- ◆ GPC: 16 个输入/输出端口及特殊功能。
- ◆ GPD: 16 个输入/输出端口及特殊功能。
- ◆ GPE: 16 个输入/输出端口及特殊功能。
- ◆ GPF: 8 个输入/输出端口及特殊功能。
- ◆ GPG: 16 个输入/输出端口及特殊功能。
- ◆ GPH: 11 个输入/输出端口及特殊功能。

对 GPIO 端口进行操作的寄存器被映射在从地址 `0x56000000` 开始的内存区域上，如表 6.7 所示。

表 6.7 GPIO 相关寄存器

寄存器	地址	读/写	描述	初始值
GPACON	0x56000000	R/W	GPA 控制寄存器	0x7FFFFF
GPADAT	0x56000004	R/W	GPA 数据寄存器	不定
GPBCON	0x56000010	R/W	GPB 控制寄存器	0x00000000
GPBDAT	0x56000014	R/W	GPB 数据寄存器	不定
GPBUP	0x56000018	R/W	GPB 禁用上拉寄存器	0x0000
GPCCON	0x56000020	R/W	GPC 控制寄存器	0x00000000
GPCDAT	0x56000024	R/W	GPC 数据寄存器	不定
GPCUP	0x56000028	R/W	GPC 禁用上拉寄存器	0x0000
GPDCON	0x56000030	R/W	GPD 控制寄存器	0x0000
GPDDAT	0x56000034	R/W	GPD 数据寄存器	不定
GPDUP	0x56000038	R/W	GPD 禁用上拉寄存器	0xF000
GPECON	0x56000040	R/W	GPE 控制寄存器	0x0000
GPEDAT	0x56000044	R/W	GPE 数据寄存器	不定
GPEUP	0x56000048	R/W	GPE 禁用上拉寄存器	0x0000
GPFCON	0x56000050	R/W	GPF 控制寄存器	0x0000
GPFDAT	0x56000054	R/W	GPF 数据寄存器	不定
GPFUP	0x56000058	R/W	GPF 禁用上拉寄存器	0x0000
GPGCON	0x56000060	R/W	GPG 控制寄存器	0x0000
GPGDAT	0x56000064	R/W	GPG 数据寄存器	不定
GPGUP	0x56000068	R/W	GPG 禁用上拉寄存器	0xF800
GPHCON	0x56000070	R/W	GPH 控制寄存器	0x0000
GPHDAT	0x56000074	R/W	GPH 数据寄存器	不定
GPHUP	0x56000078	R/W	GPH 禁用上拉寄存器	0xF800

GPA 组的端口与其他端口有所不同，这些端口只能用于输出或另外一种特殊功能（如一些控制信号、地址信号等），因此 GPACON 寄存器中的每一位控制相应的一个管脚，共有 23 个有效位。当某个位的值为 0 时，相应的管脚用于输出，为 1 时表示用于特殊功能。当管脚用于输出时，可以通过 GPADAT 寄存器设置它们的输出电平。

GPB 到 GPH 组的端口有相同点，即它们都有输入、输出和另外一种到两种特殊功能，因此 GPxCON (x 为 B~H) 寄存器中的每两位才能控制一个端口的功能。这两位值是 00 时表示相应的管脚用于输入，是 01 时表示相应的管脚用于输出，是 10 时表示用于第一种特殊功能，是 11 时表示用于第二种特殊功能或者无意义。当这些端口用于输入输出时，通过读写 GPxDAT (x 为 B~H) 寄存器可以检测或控制相应的管脚电平。

GPB 到 GPH 组中的大部分端口在芯片内部有上拉功能，通过 GPxUP (x 为 B~H) 寄存器的相应比特位可以控制上拉功能的启用与否。当某个位的值为 0 时，相应的管脚被上拉；当值为 1 时，对管脚的上拉被禁用。有些端口没有上拉功能，则相应的比特位设置无效。

以上所述是 S3C2410A 处理器 GPIO 端口的一些规律性的内容。对于 GPIO 端口还有一些其

他的配置寄存器,尤其是对其特殊功能的配置。限于篇幅,这里不再一一列出,使用时请详细参考 S3C2410A 芯片手册。

由于在前面几节的例程中已经涉及到了对 GPIO 端口的使用,这里我们不再给出新的例程。

6.5 UART 控制器及串口通信应用

UART 一般称为串口,是嵌入式处理器上最常见的资源,一般用于输出系统的控制运行信息,可作为一个控制台使用。因为串口通信的协议比较简单,不需要复杂的驱动,所以一般在系统启动早期就尽快地启用串口功能,以便进行观测和调试。因此,串口控制器在现在的嵌入式处理器上仍然是一个非常重要的接口。

在本节中,我们将首先对串口通信的硬件和软件原理进行简单的介绍,然后介绍 S3C2410A 处理器的 UART 控制器,最后给出串口通信的例程。

6.5.1 UART 通信原理

UART 是一个典型的异步通信方式,与同步通信不同,通信的双方有各自独立的工作时钟,实际的传输速度需由双方事先约定,或者采用流控方式进行同步。

在这种异步通信方式中,数据是以帧的方式传输的。当传输线空闲(没有数据在传输)时处于逻辑上的 1 状态,而每帧以逻辑上的 0 作为起始位,逻辑上的 1 作为停止位,一般有一个起始位,一个或多个停止位。每帧可传输 5~8 个比特的数据以及 1 个比特的奇偶校验位。奇偶校验位的有无及取值可由通信双方约定。通信中不同数据帧之间的时间间隔是不固定的,而在同一数据帧中的每个比特具有相同的传输时间。一帧数据传输完毕后可以继续传输下一帧数据,也可以持续高电平,这时实际上传输线将处于空闲状态。

由于发送起始位时,传输线上的状态由逻辑 1 变为逻辑 0,接收端因此可以检测到一个数据帧的开始,然后按照一个既定的速度和帧格式接收数据。当一帧结束时发送停止位,使传输线恢复为逻辑 1 的状态。传送时传输线的状态变化如图 6.3 所示。



图 6.3 UART 异步通信时序

UART 通信协议对各个数据位的具体规定如下。

- ◆ 起始位：先发出一个逻辑 0 信号,表示传输字符的开始。
- ◆ 有效数据位：紧接在起始位之后。数据位的个数可以是 4~8 个,从最低位开始传送,靠收发端的工作时钟定位。
- ◆ 奇偶校验位：数据位加上这一位后,使得 1 的位数为偶数(偶校验)或奇数(奇校验)个,接收端以此来校验数据传送的正确性。
- ◆ 停止位：它是一个数据的结束标志,可以是 1 位、1.5 位或 2 位长度的逻辑 1 信号。

◆ 空闲位：处于逻辑 1 状态，表示当前线路上没有数据在传送。

UART 通信中一个常用的参数是波特率，它是每秒钟传送的二进制位数，与收发双方的工作时钟有关，可以反映数据的最大传送速度。

6.5.2 RS-232C 串行接口标准

RS-232C 是由 EIA (Electronic Industries Alliance, 电子工业协会) 制定的串行通信接口标准，用于 DTE (Data Terminal Equipment, 数据终端设备) 与 DCE (Data Circuit-terminating Equipment, 数据通信设备) 之间的通信，如计算机和调制解调器之间，包括通信协议及接口连接标准。

具体来说，在硬件接口上，RS-232C 是一种标准的 D 型插座，采用 25 芯引脚或 9 芯引脚的连接器，实际上在 25 芯插座上也只使用了 9 芯。因此，最常用的就是 9 芯插座，如图 6.4 所示。

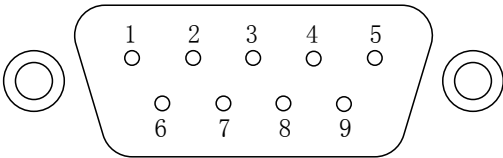


图 6.4 RS-232C 9 芯插座引脚排列

DTE 端 9 个引脚的定义如表 6.8 所示。

表 6.8 RS-232C 插座引脚定义

编号	名称	作用
1	DCD	载波检测，输入，表示 DCE 已与外部线路连接好
2	RXD	接收线，输入，用于接收数据
3	TXD	发送线，输出，用于发送数据
4	DTR	DTE 就绪，输出
5	GND	地
6	DSR	DCE 就绪，输入
7	RTS	请求发送，输出，通过此引脚通知 DCE 要求发送数据
8	CTS	清除发送
9	RI	振铃指示，输入，若 DCE 接到交换台送来的振铃呼叫信号，就发出该信号来通知 DTE

RS-232C 采用 -3V~-15V 的电平表示逻辑上的 1，而 +3V~+15V 的电平表示逻辑上的 0，与一般嵌入式处理器的 TTL 电平或 CMOS 电平不一致，因此在进行电路设计时需要增加接口转换芯片。

RS-232C 标准定义时，主要的目的是用来连接计算机终端设备和 Modem 这类通信设备，如图 6.5 所示。通过这种方式可以实现两台远程计算机间的通信。

实际上，我们在进行嵌入式开发时，主机与目标机也是通过 RS-232C 接口连接的，这种应用已不再拘泥于上述的模式，通常只用到接口中的 RXD, TXD 和 GND 连线，其余的连线均可以悬

空。HY2410A 开发板采用的就是这种连线方式。



图 6.5 RS-232C 协议的应用

6.5.3 UART 控制寄存器

S3C2410A 处理器提供 3 个独立的 UART 通信接口，并且都可以工作于中断或 DMA 方式。

在中断方式下，当 UART 单元能够发送数据时，会向 CPU 发出中断，请求发送；而接收到数据时也向 CPU 发送中断，表示接收到数据。UART 单元还支持 FIFO 的工作方式，能够通过缓存一定量的数据，减少数据传输时的中断次数，提高传输效率。

在 DMA 方式下，UART 单元可以与系统内存间直接传输数据，传输过程中不需要 CPU 的介入。

S3C2410A 处理器的 UART 单元由收发器、波特率发生器及控制器组成。波特率发生器可产生高达 230.4 kb/s 的波特率，产生的波特率可由软件控制。通过控制器可以改变波特率、设置停止位为 1 个或 2 个、数据位数为 5~8 位以及校验位的设置，并且还可以控制是否通过红外接口进行发送与接收。

S3C2410A 处理器的每个 UART 单元都有 16 字节的发送 FIFO 和 16 字节的接收 FIFO，这样在中断或 DMA 传输时可以避免频繁地对处理器进行中断，以改善串口通信的效率。

S3C2410A 处理器的 UART 控制器相关的寄存器被映射在从地址 0x50000000 开始的内存区域上，如表 6.9 所示。

表 6.9 UART 相关寄存器

寄存器	地址	读/写	描述	初始值
ULCON0	0x50000000	R/W	UART0 线路控制寄存器	0x00
UCON0	0x50000004	R/W	UART0 控制寄存器	0x00
UFCON0	0x50000008	R/W	UART0 FIFO 控制寄存器	0x00
UMCON0	0x5000000C	R/W	UART0 Modem 控制寄存器	0x00
UTRSTAT0	0x50000010	R	UART0 发送接收状态寄存器	0x06
UERSTAT0	0x50000014	R	UART0 错误状态寄存器	0x00
UFSTAT0	0x50000018	R	UART0 FIFO 状态寄存器	0x00
UMSTAT0	0x5000001C	R	UART0 Modem 状态寄存器	0x00
UTXH0	0x50000020 (小端) 0x50000023 (大端)	W	UART0 发送缓冲寄存器	无
URXH0	0x50004024 (小端) 0x50004027 (大端)	R	UART0 接收缓冲寄存器	无
UBRDIV0	0x50000028	R/W	UART0 波特率除数寄存器	无
ULCON1	0x50004000	R/W	UART1 线路控制寄存器	0x00
UCON1	0x50004004	R/W	UART1 控制寄存器	0x00



(续表)

寄存器	地址	读/写	描述	初始值
UFCON1	0x50004008	R/W	UART1 FIFO 控制寄存器	0x00
UMCON1	0x5000400C	R/W	UART1 Modem 控制寄存器	0x00
UTRSTAT1	0x50004010	R	UART1 发送接收状态寄存器	0x06
UERSTAT1	0x50004014	R	UART1 错误状态寄存器	0x00
UFSTAT1	0x50004018	R	UART1 FIFO 状态寄存器	0x00
UMSTAT1	0x5000401C	R	UART1 Modem 状态寄存器	0x00
UTXH1	0x50004020 (小端) 0x50004023 (大端)	W	UART1 发送缓冲寄存器	无
URXH1	0x50004024 (小端) 0x50004027 (大端)	R	UART1 接收缓冲寄存器	无
UBRDIV1	0x50004028	R/W	UART1 波特率除数寄存器	无
ULCON2	0x50008000	R/W	UART2 线路控制寄存器	0x00
UCON2	0x50008004	R/W	UART2 控制寄存器	0x00
UFCON2	0x50008008	R/W	UART2 FIFO 控制寄存器	0x00
UTRSTAT2	0x50008010	R	UART2 发送接收状态寄存器	0x06
UERSTAT2	0x50008014	R	UART2 错误状态寄存器	0x00
UFSTAT2	0x50008018	R	UART2 FIFO 状态寄存器	0x00
UTXH2	0x50008020 (小端) 0x50008023 (大端)	W	UART2 发送缓冲寄存器	无
URXH2	0x50008024 (小端) 0x50008027 (大端)	R	UART2 接收缓冲寄存器	无
UBRDIV2	0x50008028	R/W	UART2 波特率除数寄存器	无

软件通过向 UART 的发送缓冲寄存器 $UTXH_n$ (n 为 0~2 的数字) 写入数据进行发送操作, 通过读取接收缓冲寄存器 $URXH_n$ (n 为 0~2 的数字) 来获取接收到的数据。发送与接收都是以字节为单位进行的, 因此只有寄存器的低 8 位有效。

S3C2410A 串口的时钟一般为 PCLK。通过设置 $UBRDIV_n$ (n 为 0~2 的数字) 寄存器, 可以得到相应的波特率, 计算公式如下:

$$UBRDIV_n = (\text{int})(PCLK/(\text{波特率} \times 16)) - 1$$

根据上述公式, 如果 PCLK 频率为 50MHz, 波特率设置为 115200b/s, 则可以配置 $UBRDIV_n$ 的值为 26。

UART 线路控制寄存器 $ULCON_n$ (n 为 0~2 的数字) 的各个比特位的具体含义如表 6.10 所示。



表 6.10 UART 线路控制寄存器说明

比特位	含义	描述
6	红外模式标志	0 = 普通模式, 1 = 红外收发模式
5 - 3	校验模式	0xx = 无校验, 100 = 奇校验, 101 = 偶校验, 110 = 校验位强制为 1, 111 = 校验位强制为 0
2	停止位长度	0 = 1 位, 1 = 2 位
1 - 0	数据位长度	00 = 5 位, 01 = 6 位, 02 = 7 位, 03 = 8 位

UART 控制寄存器 UCON_n (*n* 为 0~2 的数字) 的各个比特位的具体含义如表 6.11 所示。

表 6.11 UART 控制寄存器说明

比特位	含义	描述
10	时钟选择	0 = PCLK, 1 = 外部时钟
9	Tx 中断类型	0 = 脉冲类型, 1 = 电平类型
8	Rx 中断类型	0 = 脉冲类型, 1 = 电平类型
7	Rx 超时使能	0 = 禁止接收超时中断, 1 = 允许接收超时中断
6	Rx 错误状态中断使能	0 = 不产生接收错误中断, 1 = 产生接收错误中断
5	环回模式选择	0 = 正常模式, 1 = 环回模式
4	保留	无
3 - 2	发送模式	00 = 禁止发送, 01 = 中断或查询模式, 10 = 使用 DMA0(仅用于 UART0)或使用 DMA3(仅用于 UART2), 11 = 使用 DMA1(仅用于 UART1)
1 - 0	接收模式	00 = 禁止接收, 01 = 中断或查询模式, 10 = 使用 DMA0(仅用于 UART0)或使用 DMA3(仅用于 UART2), 11 = 使用 DMA1(仅用于 UART1)

UART FIFO 控制寄存器 UFCON_n (*n* 为 0~2 的数字) 的各个比特位的具体含义如表 6.12 所示。

表 6.12 UART FIFO 控制寄存器说明

比特位	含义	描述
7 - 6	Tx FIFO 触发水平	00 = 空, 01 = 4 字节, 10 = 8 字节, 11 = 12 字节
5 - 4	Rx FIFO 触发水平	00 = 4 字节, 01 = 8 字节, 10 = 12 字节, 11 = 满
3	保留	无
2	Tx FIFO 复位	0 = 无操作, 1 = 清空 Tx FIFO
1	Rx FIFO 复位	0 = 无操作, 1 = 清空 Rx FIFO
0	FIFO 使能标志	0 = 不使用 FIFO, 1 = 使用 FIFO

UART Modem 控制寄存器 UMCON_n (*n* 为 0~1 的数字) 的各个比特位的具体含义如表 6.13 所示。注意 UART2 不支持这些功能。



表 6.13 UART Modem 控制寄存器说明

比特位	含义	描述
7 – 5	保留	无
4	AFC 控制	0 = 禁用 AFC, 1 = 使用 AFC
3 – 1	保留	无
0	请求发送	0 = nRTS 高电平（不激活）, 1 = nRTS 低电平（激活）

UART 发送接收状态寄存器 $UTRSTAT_n$ (n 为 0~2 的数字) 的各个比特位的具体含义如表 6.14 所示。注意这些寄存器都是只读的，并且在使用 FIFO 的模式下应该依据 FIFO 状态寄存器的值进行判断而不能再依据此寄存器。

表 6.14 UART 发送接收状态寄存器说明

比特位	含义	描述
2	发送器空	0 = 发送器非空, 1 = 发送器空
1	发送缓冲区空	0 = 缓冲区非空, 1 = 缓冲区空
0	接收缓冲区数据就绪	0 = 无数据, 1 = 接收到数据

UART 错误状态寄存器 $UERSTAT_n$ (n 为 0~2 的数字) 的各个比特位的具体含义如表 6.15 所示。注意这些寄存器都是只读的，并且对它们的读取操作会使之自动清零。

表 6.15 UART 错误状态寄存器说明

比特位	含义	描述
3	保留	无
2	帧错误	0 = 无错误, 1 = 有错误
1	保留	无
0	溢出错误	0 = 无错误, 1 = 有错误

UART FIFO 状态寄存器 $UFSTAT_n$ (n 为 0~2 的数字) 的各个比特位的具体含义如表 6.16 所示。注意这些寄存器都是只读的。

表 6.16 UART FIFO 状态寄存器说明

比特位	含义	描述
9	发送 FIFO 满	0 = 未滿, 1 = 滿
8	接收 FIFO 满	0 = 未滿, 1 = 滿
7 – 4	发送 FIFO 计数	发送 FIFO 中的数据个数
3 – 0	接收 FIFO 计数	接收 FIFO 中的数据个数

UART Modem 状态寄存器 $UMSTAT_n$ (n 为 0~2 的数字) 的各个比特位的具体含义如表 6.17 所示。注意这些寄存器都是只读的。



表 6.17 UART Modem 状态寄存器说明

比特位	含义	描述
4	CTS 状态变化指示	0 = nCTS 状态无变化, 1 = nCTS 状态有变化
3 - 1	保留	无
0	CTS 信号指示	0 = CTS 信号未激活, 1 = CTS 信号激活

6.5.4 串口通信应用例程

在前面几节的例程中, 我们利用 U-boot 提供的 `printf` 函数输出调试信息, 实际上就是在通过串口与主机通信。在这个例程中, 我们将编程直接操作串口, 以实现基本的输入输出功能。

例程的主要部分是一个 C 语言源文件, 内容如下:

```
/* 文件名: c_uart.c */
/* 说明: 串口通信例程 */

#define S3C24XX_PA_UART 0x50000000 /* UART 寄存器基址 */

#define S3C24XX_PA_UART0 (S3C24XX_PA_UART + 0x0000) /* UART0 寄存器基址 */
#define S3C24XX_PA_UART1 (S3C24XX_PA_UART + 0x4000) /* UART1 寄存器基址 */
#define S3C24XX_PA_UART2 (S3C24XX_PA_UART + 0x8000) /* UART2 寄存器基址 */

typedef volatile unsigned char S3C24XX_REG8;
typedef volatile unsigned int S3C24XX_REG32;

/* 通过以下结构体映射到 UART 的各个寄存器 */
typedef struct {
    S3C24XX_REG32 ULCON;
    S3C24XX_REG32 UCON;
    S3C24XX_REG32 UFCON;
    S3C24XX_REG32 UMCON;
    S3C24XX_REG32 UTRSTAT;
    S3C24XX_REG32 UERSTAT;
    S3C24XX_REG32 UFSTAT;
    S3C24XX_REG32 UMSTAT;
    S3C24XX_REG8 UTXH;
    S3C24XX_REG8 res1[3];
    S3C24XX_REG8 URXH;
    S3C24XX_REG8 res2[3];
    S3C24XX_REG32 UBRDIV;
} S3C24XX_UART;

/* 输出一个字符 */
int s_putc(char c)
{
    S3C24XX_UART *const uart = (S3C24XX_UART *const)S3C24XX_PA_UART0;
    while (!(uart->UTRSTAT & 0x2)); /* 等待直到发送缓冲区空 */
    uart->UTXH = c; /* 发送 */
    if (c == '\n') s_putc('\r'); /* 如果是换行符则再发送一个回车符 */
}
```



```

    return (int)c;
}

/* 输入一个字符 */
int s_getc(void)
{
    char c;
    S3C24XX_UART *const uart = (S3C24XX_UART *const)S3C24XX_PA_UART0;
    while (!(uart->UTRSTAT & 0x1)); /* 等待直到接收缓冲区有数据 */
    c = uart->URXH; /* 接收 */
    if (c == '\r') c = '\n'; /* 将回车符替换为换行符 */
    s_putc(c); /* 输出这个字符 (回显) */
    return (int)c;
}

/* 输出一个字符串 */
const char *s_puts(const char *s)
{
    const char *p = s;
    while (*p) {
        s_putc(*p++);
    }
    return s;
}

/* 输入一个字符串 (以回车结尾) */
char *s_gets(char *buf)
{
    int c;
    char *p = buf;
    do {
        c = s_getc();
        *p++ = (char)c;
    } while ((char)c != '\n');
    p[-1] = '\0'; /* 加上字符串结束标志 */
    return buf;
}

/* 测试以上函数 */
void test(void)
{
    char buf[60];
    s_puts("Please input something:");
    s_gets(buf);
    s_puts("String is: ");
    s_puts(buf);
    s_putc('\n');
}

```

这个源码中实现了字符输出、输入以及字符串输出、输入的函数。实现的方式是直接读写串口



寄存器。因为 UART 控制器的各个寄存器对应的内存地址是连续的，因此这里使用了一种技巧，即定义一个结构体，它的各个成员与寄存器一一对应，将结构体变量的首地址指向第一个寄存器的地址，这样结构体的所有成员就与各个寄存器的地址有了对应关系。

由于使用的是 UART0，这个串口在 U-boot 启动时已经配置好，所以不需要再进行配置，可以直接开始读写操作。U-boot 中设置串口的代码如下：

```
void serial_setbrg (void)
{
    S3C24X0_UART * const uart = S3C24X0_GetBase_UART(UART_NR);
    int i;
    unsigned int reg = 0;
    /* 计算波特率除数寄存器的值 */
    reg = get_PCLK() / (16 * gd->baudrate) - 1;
    /* 启用 FIFO, 清空 Tx/Rx FIFO */
    uart->UFCON = 0x07;
    uart->UMCON = 0x0;
    /* 普通模式, 无校验, 1 位停止位, 8 比特数据 */
    uart->ULCON = 0x3;
    /* 发送为电平类型中断, 接收为脉冲类型中断,
     * 禁用超时中断, 使能接收错误中断
     */
    uart->UCON = 0x245;
    uart->UBRDIV = reg;
#ifdef CONFIG_HWFLOW /* 如果需要硬件流控制 */
    uart->UMCON = 0x1; /* 启用 RTS */
#endif
    for (i = 0; i < 100; i++); /* 延时 */
}
```

同样，这个例程需要一小段汇编程序作为引子跳转到测试函数处，内容如下：

```
@ 文件名: uart.S
@ 说明: 串口通信例程

.global _start
.text
.extern test
_start:
    b     test
.end
```

6.6 NAND Flash 芯片与控制器

Flash 存储芯片，俗称闪存，因其具有非易失性、可擦除性、可重复编程及高密度、低功耗等特点，广泛地应用于手机、数码相机、笔记本电脑等产品。

根据制造技术不同，可将 Flash 存储芯片分为两类：NOR Flash 和 NAND Flash。NOR Flash 的传输效率很高，但写入和擦除速度较慢；相比之下，NAND Flash 具有容量大、写速度快、芯片面



积小、单元密度高、擦除速度快、成本低等优点，这些优点使其成为高密度数据存储的理想解决方案。从产品应用选择来讲，NAND Flash 更适合嵌入式系统的大容量数据存储使用。

从 NOR Flash 读取数据的方式与从 RAM 读取数据相近，有分离的地址线和数据线，只要提供数据的地址，数据总线就可以正确地给出数据。基于以上原因，多数处理器可以将 NOR Flash 作为 XIP (Execute In Place, 原地执行) 内存使用，这意味着储存在 NOR Flash 上的程序不需要复制到 RAM 中就可以直接执行。虽然从 NOR Flash 读取数据很方便并且速度快，但与 NAND Flash 相比，NOR Flash 的写入速度一般来说会慢很多。

NAND Flash 的存取方式类似于硬盘，数据的存取以固定大小的区块进行，这些区块称为页，一般来说页的尺寸为 512, 2048 或 4096 字节。另外各个页之间会有一些额外的空间用于储存检错与纠错的校验和。处理器不能直接对 NAND Flash 进行数据存取操作，必须通过 NAND Flash 控制器进行。

向 Flash 写入数据前必须将原有的内容擦除，擦除操作只能以块为单位进行，每个块由几个页构成。一个块的擦除次数是有限制的，通常称为 Flash 的擦除寿命。

本节的主要内容是介绍 NAND Flash 芯片的基本结构和 S3C2410A 处理器的 NAND Flash 控制器，并通过实例来说明如何对 HY2410A 开发板所使用的 NAND Flash 芯片进行操作。

6.6.1 NAND Flash 的访问

NAND Flash 中的最小存储单元称为 cell，用来保存一个比特的数据。这些 cell 以 8 个或者 16 个为单位组合在一起，形成所谓的 bit line，用于保存一个 8 位数据或 16 位数据。

这些 bit line 会进一步组成页。NAND Flash 有多种结构，HY2410A 开发板使用的是三星公司的 K9F1208 芯片，它的一页由 512 字节的主数据区和 16 字节的额外数据区组成。而每 32 个页又形成一个块，芯片内总共有 4096 个块，故这款芯片的容量为 66M 字节，但其中 2M 字节是用来保存 ECC 校验码等额外数据的，因此实际中可使用的有效容量为 64M 字节。

K9F1208 芯片使用一个 26 位的整数来表示数据的地址，地址可划分为两个部分：页地址和页内偏移地址。页地址是其高 17 位，可以定位到页；页内偏移地址是其低 9 位，可以定位到字节。页地址的高 12 位实际上就是块的编号，可称为块地址。理论上来说，我们将这个 26 位的地址传送给芯片就可以定位到任意一个字节。

实际上，这款芯片在访问上还有一个特殊的设计，即将一个页分为两个部分：上半页和下半页。具体对哪个半页访问取决于所用的读取命令，而在传递地址时只需要页内偏移地址的低 8 位，称为列地址。整个地址的结构如图 6.6 所示。

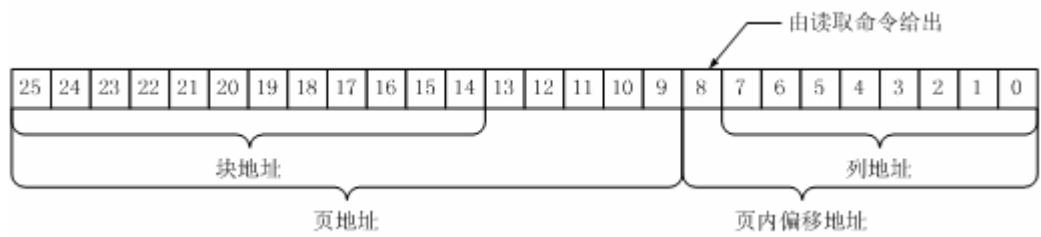


图 6.6 NAND Flash 的地址

对 NAND Flash 进行访问的方式一般是先发送命令，再发送地址，然后进行数据的存取。对



于 K9F1208 芯片来说,不管是命令还是数据和地址,都是通过芯片的一个 8 位 I/O 口来传送的。如表 6.18 所示是 K9F1208 芯片所支持的命令字。

表 6.18 K9F1208 命令字

功能	第 1 周期	第 2 周期	第 3 周期
从上半页开始读	0x00	无	无
从下半页开始读	0x01	无	无
从额外数据区开始读	0x50	无	无
读 ID	0x90	无	无
复位	0xFF	无	无
写入	0x80	0x10	无
写入(空操作)	0x80	0x11	无
回写	0x00	0x8A	0x10
回写(空操作)	0x03	0x8A	0x11
块擦除	0x60	0xD0	无
读状态	0x70	无	无
读多平面状态	0x71	无	无

可以看出,读操作只有 1 个周期的命令字,发送完命令字之后发送地址,然后就可以连续读出数据。写操作和擦除操作的命令字则一般有两个周期,首先发送第一周期的命令字,然后发送地址和数据,最后发送第二周期的命令字,写操作才真正开始。而回写操作则有三个周期的命令字,首先发送第一周期的命令字,然后发送源地址,再发送第二周期的命令字,然后发送目标地址,最后发送第三周期的命令字,数据将从源地址被读出并写入目标地址。

此外, K9F1208 芯片还支持所谓的多平面写入,即连续向四个不同的地址发送写入命令,最后同时启动写操作,这种操作可以提高写入的速度,它是由写入的空操作命令与写入命令配合来实现的,具体的操作时序请参考 K9F1208 芯片手册。

下面以读操作为例说明对 K9F1208 芯片的整个操作过程。首先,处理器的 NAND Flash 控制器发送命令字,这时 CLE 信号高电平有效,表示 8 位 I/O 线上给出的是命令字。然后控制器发送地址,这时 ALE 信号高电平有效,8 位 I/O 线上连续进行 4 次写操作,将地址送出。随后,芯片需要一点时间准备数据,它给出的 RnB 信号为低电平表示芯片在忙状态,处理器必须进行等待直到 RnB 恢复为高电平,然后开始读取数据。需要指出的是,芯片给出的不仅是指定地址处的数据,只要处理器不断地进行读取,就可以得到整页内的所有数据,包括额外数据区。

不管是命令字还是地址,对芯片的写入都由 nWE 信号控制,每次 nWE 信号由高电平跳变为低电平,芯片就接收一次数据。读出则由 nRE 信号控制,每次 nRE 信号由高电平跳变到低电平,芯片就给出一个新的数据。

当 RnB 状态指示为忙时,芯片一般是不接受命令的,只有复位操作和读状态操作可以进行。

顺便提一句,额外数据区中的数据可以被基于 NAND Flash 的文件系统使用,比如嵌入式文件系统 YAFFS 就使用了这个额外数据区以保存一些文件系统的状态信息。

如表 6.19 所示是传送地址时地址的各个比特位与 8 位 I/O 线的对应关系。从中可以看出,地址的第 9 位 A8 是不需要传送的,实际上隐含在读命令中了。当然,在进行其他操作时,地址



的传送不一定按照这种方式，比如擦除时不需要给出列地址，因此地址的发送只需要 3 个周期，而读 ID 和状态时根本不需要地址。

表 6.19 地址的传送方式

周期	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O 5	I/O 6	I/O 7
第 1 周期	A0	A1	A2	A3	A4	A5	A6	A7
第 2 周期	A9	A10	A11	A12	A13	A14	A15	A16
第 3 周期	A17	A18	A19	A20	A21	A22	A23	A24
第 4 周期	A25	0	0	0	0	0	0	0

6.6.2 NAND Flash 控制器

下面我们具体介绍 S3C2410A 处理器的 NAND Flash 控制器的配置和使用方法。与 NAND Flash 控制器相关的寄存器被映射在从地址 0x4E000000 开始的内存区域上，如表 6.20 所示。

表 6.20 NAND Flash 控制器相关的寄存器

寄存器	地址	读/写	描述	初始值
NFCONF	0x4E000000	R/W	NAND Flash 配置寄存器	无
NFCMD	0x4E000004	R/W	NAND Flash 命令设置寄存器	0x00
NFADDR	0x4E000008	R/W	NAND Flash 地址设置寄存器	0x00
NFDATA	0x4E00000C	R/W	NAND Flash 数据寄存器	无
NFSTAT	0x4E000010	R	NAND Flash 状态寄存器	无
NFECC	0x4E000014	R	NAND Flash ECC 寄存器	无

NAND Flash 配置寄存器的各个比特位的具体含义如表 6.21 所示。

表 6.21 NAND Flash 配置寄存器说明

比特位	名称	描述	初始值
15	控制器使能	0 = 禁用控制器，1 = 使用控制器	0
14 - 13	保留	无	无
12	ECC 校验控制	0 = 无操作，1 = 初始化 ECC 校验	0
11	NAND Flash 片选	0 = 片选信号有效，1 = 片选信号无效 (处理器重启后，片选信号为无效)	无
10 - 8	TACLS	CLE 或 ALE 信号的建立时间设置， 建立时间为 $HCLK \times (TACLS + 1)$	0
7	保留	无	无
6 - 4	TWRPH0	写信号保持时间设置， 保持时间为 $HCLK \times (TWRPH0 + 1)$	0
3	保留	无	无
2 - 0	TWRPH1	CLE 或 ALE 撤销时间设置， 撤销时间为 $HCLK \times (TWRPH1 + 1)$	0



其中 TACLS, TWRPH0 和 TWRPH1 参数的含义如图 6.7 所示, 图中是 TACLS = 0, TWRPH0 = 1, TWRPH1 = 0 的情形, 这些值的设置与 NAND Flash 芯片的响应时间有关, 必须参考芯片的手册以计算出合适的值。如果设置的时间过短, 则芯片来不及响应, 可能造成数据传送错误。

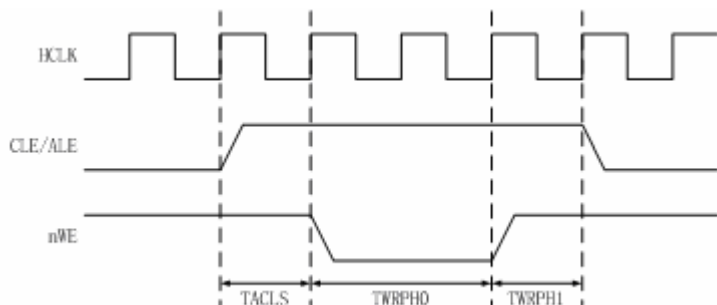


图 6.7 CLE/ALE 与 nWE 信号时序

向 NAND Flash 发送命令字可通过向命令设置寄存器 NFCMD 写入数据来实现, 数据的低 8 位为有效值。

向 NAND Flash 发送地址可通过向地址设置寄存器 NFADDR 写入数据来实现, 数据的低 8 位为有效值, 因此传送一个完整的地址时必须由软件将其分为多次写入。

向 NAND Flash 写入数据或从 NAND Flash 读取数据可通过访问数据寄存器 NFDATA 来实现, 寄存器的低 8 位为有效值。

状态寄存器 NFSTAT 则用来检测 NAND Flash 芯片的状态, 它的最低位表示 RnB 信号, 其余位均为保留。当 RnB 的值为 0 时, 表示芯片忙; 值为 1 时表示芯片就绪, 可以进行操作。

ECC 寄存器 NFECCE 的低 24 位则表示读写一页数据时由 ECC 模块计算好的 ECC 值。

下面是进行读操作时 NAND Flash 控制器的设置过程。

- step 1** 设置控制器访问 NAND Flash 的时序及其他相关比特位, 如片选、控制器使能等。
- step 2** 向 NFCMD 寄存器写入命令字。
- step 3** 向 NFADDR 寄存器分多次写入地址。
- step 4** 根据 NFSTAT 寄存器判断 NAND Flash 芯片是否就绪。
- step 5** 当芯片就绪时, 读取 NFDATA 寄存器的内容以获取数据。

6.6.3 NAND Flash 控制器编程实例

这里我们编写一段程序用来从 NAND Flash 读取数据到内存。例程的主要部分是 C 语言程序, 内容如下:

```
/* 文件名: c_nandop.c */
/* 说明: NAND Flash 控制器例程 */

#define __REGb(x) (*(volatile unsigned char *)(x))
#define __REGi(x) (*(volatile unsigned int *)(x))

#define NF_BASE      0x4e000000
#define NFCONF       __REGi(NF_BASE + 0x0)
```

```

#define NFCMD        __REGB(NF_BASE + 0x4)
#define NFADDR        __REGB(NF_BASE + 0x8)
#define NFDATA        __REGB(NF_BASE + 0xc)
#define NFSTAT        __REGB(NF_BASE + 0x10)

#define BUSY        1
#define TACLS        0
#define TWRPH0        3
#define TWRPH1        0

#define NAND_SECTOR_SIZE    512
#define NAND_BLOCK_MASK    (NAND_SECTOR_SIZE - 1)

/* 等待 NAND Flash 芯片就绪 */
inline void wait_idle(void)
{
    int i;
    while (!(NFSTAT & BUSY))
        for (i = 0; i < 10; i++); /* 延时 */
}

/* 复位 NAND Flash 芯片 */
static void nand_reset(void)
{
    /* NAND Flash 片选使能 */
    NFCONF &= ~0x800;
    /* NAND Flash 复位命令 */
    NFCMD = 0xff;
    wait_idle();
    /* NAND Flash 片选禁止 */
    NFCONF |= 0x800;
}

/* 初始化 NAND Flash 控制器并复位芯片 */
void nand_init(void)
{
    /* 设置时序、使能控制器、初始化 ECC、禁止片选 */
    NFCONF = (1<<15)|(1<<12)|(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0);
    /* 复位 NAND Flash */
    nand_reset();
}

/* 从地址 start_addr 开始, 读 size 长度的数据, 到缓冲区 buf 中 */
int nand_read_ll(unsigned char *buf, unsigned long start_addr, int size)
{
    int i, j;
    /* 检查开始地址及提取的数据长度是否在页边界 */
    if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK)) {
        return -1;
    }
    /* 使能 NAND Flash 片选 */

```

```

    NFCONF &= ~0x800;
    for (i = 0; i < 10; i++); /* 延时 */
    for (i = start_addr; i < (start_addr + size);) {
        /* 发送读命令, 从上半区开始读 */
        NFCMD = 0x00;
        /* 发送地址 */
        NFADDR = i & 0xff;
        NFADDR = (i >> 9) & 0xff;
        NFADDR = (i >> 17) & 0xff;
        NFADDR = (i >> 25) & 0xff;
        /* 等待芯片送出数据 */
        wait_idle();
        /* 读页数据, 忽略额外数据, 然后读下一页 */
        for (j = 0; j < NAND_SECTOR_SIZE; j++, i++) {
            *buf = (NFDATA & 0xff);
            buf++;
        }
    }
    /* 禁用 NAND Flash 片选 */
    NFCONF |= 0x800;
    return 0;
}

/* 测试读操作 */
void nand_read(unsigned char *buf, unsigned long start_addr, int size)
{
    int i, ret;
    nand_init();
    ret = nand_read_ll(buf, start_addr, size);
    if (ret) {
        printf("Read error!\n");
    } else {
        /* 输出读到的数据 */
        for (i = 0; i < size; i++) {
            printf("%02X ", buf[i]);
            if ((i & 0xF) == 0xF) printf("\n");
        }
    }
}
}

```

下面是程序的入口汇编代码:

```

@ 文件名: nandop.S
@ 说明: NAND Flash 控制器例程

.global _start
.text
.extern nand_read
_start:
    ldr    r0, =0x30010000 @ 缓冲区地址
    mov    r1, #0 @ Flash 起始地址

```



```
mov    r2, #512 @ 长度
b      nand_read
.end
```

在这段汇编代码中，我们通过 r0, r1 和 r2 寄存器向 nand_read 函数传递了参数，以读取 NAND Flash 的第 0 页。



第 1 章 U-boot 源码分析与移植

bootloader 是嵌入式系统中一个非常重要的部分，它为初始的硬件环境提供一个可操作的软件环境，驱动一些关键的硬件接口，这样，其他的系统软件（比如操作系统）可以加载上来，构成更便于用户开发的软件环境。

U-boot 是德国 DENX 小组开发的用于多种嵌入式 CPU 的开放源代码 **bootloader** 程序。U-boot 是在 **ppcboot** 以及 **armboot** 的基础上发展而来的，现已非常成熟和稳定，并提供较完备的文档，因此在许多嵌入式系统的开发中被采用。

当前，U-boot 支持的处理器架构包括 ARM, x86, MIPS, NIOS, powerPC 等，支持的操作系统包括 Linux, NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS 等。在开源领域，U-boot 对 Linux 系统的支持最完善，因此是嵌入式 Linux **bootloader** 的最佳选择。

本章首先讨论 **bootloader** 在整个系统中充当的角色，然后详细研究 U-boot 软件，对其架构进行分析，并移植到 HY2410A 开发板平台上来。

本章的目的在于使读者了解 U-boot 后，能够根据不同的硬件方案，修改 U-boot 软件以适合自己所需。

7.1 **bootloader** 的概念

bootloader 是系统上电后最初加载运行的代码，对嵌入式系统来说，这是一个非常重要的组成部分。能够运行程序对任何开发板或计算机设备来讲都是最关键的一步，而即使一个很简单的程序要运行，也必须事先初始化很多最基本的硬件单元。各种架构的处理器都有自己加电以后最开始的执行过程，比如开始时的指令地址，指令是原地执行还是从外部存储器中搬移进内存再执行，开始的代码应该放在什么位置。这个最初始的代码就是 **bootloader** 的一部分。

总之，**bootloader** 提供了处理器上电复位后最开始需要执行的初始化代码。我们熟悉的 PC 机中的引导程序一般由 BIOS 开始执行，然后读取硬盘中位于 MBR (Main Boot Record, 主引导记录) 中的 **bootloader** (例如 LILO 或者 GRUB)，并进一步引导操作系统的启动。然而在嵌入式系统中通常没有像 BIOS 那样的固件程序（部分嵌入式 CPU 有），因此整个系统的加载启动任务就完全由 **bootloader** 来完成。

简单地说，**bootloader** 就是在操作系统内核运行前就运行的一段小程序。这段小程序可以初始化必要的硬件设备，将系统的软硬件环境带到一个合适的状态，并从外部存储器或通过网络等通信方式加载内核，创建内核需要的一些信息并将这些信息通过相关机制传递给内核，最终调用操作系统内核。

一个嵌入式 Linux 系统从软件的角度看通常可以分为四个部分：**bootloader**、Linux 内核、根文件系统及用户的应用程序。**bootloader** 处于系统的最底层，运行于系统启动的最初阶段。

bootloader 是依赖于硬件而实现的，特别是在嵌入式系统中。不同的体系结构要求的 **bootloader** 是不同的。除了体系结构，**bootloader** 还依赖于具体的嵌入式设备的配置，也就是说，对于两个不同的嵌入式设备，即使它们使用相同的处理器，运行在其中一个设备上的 **bootloader** 也不一定能够在另一个设备上运行。

bootloader 最主要的功能是加载与引导内核映像。但是随着嵌入式系统的发展，**bootloader** 已经逐渐在基本功能的基础上，增加了更多的对硬件的支持，扩展了更多的功能。这些增加的功能可以大大方便开发人员进行调试操作。从这个层面上看，功能扩展后的 **bootloader** 可以看成是一个微小的系统级的代码包。

本节的主要内容是介绍 **bootloader** 的启动过程、操作模式，以及 ARM 架构下 **bootloader** 的主要特点。

7.1.1 **bootloader** 的启动过程

bootloader 的启动过程可以是单阶段的，也可以是多阶段的。多阶段的 **bootloader** 一般比单阶段的 **bootloader** 提供更为复杂的功能，以及更好的可移植性。从固态存储设备上启动的 **bootloader** 大多数是二阶段的启动过程，比如 U-boot。

下面将以适用于 ARM 处理器的 U-boot 为例，对两个阶段中 **bootloader** 所做的操作进行简要说明。

一、第一阶段

在启动的第一阶段，**bootloader** 执行最基本的硬件初始化操作，如关闭中断、关闭看门狗以避免处理器被复位，以及关闭 MMU 功能、关闭处理器缓存（数据缓存一定要关闭，指令缓存可以打开）、设置系统时钟、初始化内存等。

这一阶段的代码通常由汇编语言编写，为了运行下一阶段的 C 语言程序还必须设置好堆栈。如果是从 NAND Flash 启动，则必须通过 NAND Flash 控制器将 **bootloder** 代码复制到内存。

二、第二阶段

第二阶段的代码一般用 C 语言编写，大体分为以下步骤。

- step 1** 初始化各种硬件设备，比如设置处理器正常工作的时钟频率、初始化串口等。
- step 2** 检测系统内存，主要是确定系统内存容量以及其地址空间信息。
- step 3** 将内核映像文件加载到内存。
- step 4** 准备内核引导参数。
- step 5** 跳转到内核的第一条指令处，开始执行内核初始化代码，控制权转移到内核代码，**bootloder** 的使命结束。

7.1.2 **bootloader** 的操作模式

大多数 **bootloader** 都包含两种不同的操作模式：“启动加载”模式和“下载”模式，这种区别对于开发人员才有意义。从最终用户的角度看，**bootloader** 的作用只是用来加载操作系统内核，并不存在所谓的“下载”工作模式。

一、“启动加载”模式

这种模式也称为“自主”模式，即 **bootloader** 从目标机上的某个固体存储设备上将操作系统加载到内存中运行，整个过程没有用户的介入。这种模式是 **bootloader** 的正常工作模式，当嵌入式系统以产品形式发布的时候，**bootloader** 必须工作在这种模式下。

二、“下载”模式

在这种模式下，目标机上的 **bootloader** 将通过串口、网络连接或者其他通信手段从主机下载文件，比如下载内核映像或根文件系统映像等。从主机下载的文件通常被保存到目标机的内存中，然后再写入到目标机上的 **Flash** 等固态存储设备中。**bootloader** 的这种工作模式通常在第一次安装内核与根文件系统时使用，或者在系统更新时使用。工作于这种模式下的 **bootloader** 通常会向它的终端用户提供一个简单的命令行接口。进行嵌入式系统调试时一般也让 **bootloader** 工作在这一模式下。

7.1.3 ARM bootloader 的特点

从上面 **bootloader** 的基本概念可以看出，**bootloader** 的设计与实现是与具体的 CPU 以及具体的硬件系统紧密相关的，要实现一个通用的 **ARM bootloader** 以适合所有的 **ARM** 处理器以及硬件系统，是不太可能的事情。另外，不同的操作系统，可能对具体的 **bootloader** 还会有另外的要求。

尽管如此，我们还是可以根据 **ARM** 的体系结构，从理论上总结出一些 **ARM** 平台上 **bootloader** 的共性。这些共性只能局限于 **bootloader** 的基本功能。因为各种 **ARM** 处理器扩展的接口不同，可以有串口、USB、以太网接口、IDE、CF 等，无法体现出共性。

对于一个运行于 **ARM** 平台的系统来说，**bootloader** 作为引导与加载内核映像的工具需要提供以下几个功能。

- ◆ 初始化内存：**bootloader** 必须能够初始化内存。
- ◆ 初始化串口：虽然系统的启动并不一定依赖于串口，但一般来说 **bootloader** 应该初始化至少一个串口，通过它与一台主机进行通信，以便进行开发、调试和维护工作。
- ◆ 创建内核参数列表：这是 **Linux** 内核所要求的，如果不给出内核参数，则内核就会使用其默认参数。
- ◆ 加载内核映像到内存：一般来说，内核映像必须在内存中运行，所以必须从其他非易失性存储介质上复制到内存。
- ◆ 启动内核镜像：让执行流程跳转到内核映像的入口。

当启动内核时，系统必须处于指定的状态，包括处理器模式、MMU 和缓存的设置、寄存器的设置等方面。

一、处理器模式

处理器应处于 **SVC** 模式，在这种特权模式下，内核才能执行所有的指令。

中断必须关闭。在异常向量表尚未初始化的情况下，如果发生中断，将导致系统崩溃。一般来说，**bootloader** 本身也没有必要支持中断的实现，这属于内核的管理范围。

二、MMU 和缓存设置

MMU 必须关闭。启动 MMU 进入保护模式是内核的工作。而 `bootloader` 本身工作在实模式下，所有对地址的操作使用的都是物理地址，不存在虚拟地址。

数据缓存必须关闭。`bootloader` 的主要功能是装载内核映像，映像数据必须真实写回内存中，不能仅放在处理器的缓存中，所以数据缓存必须关闭。

指令缓存可以打开。一般情况下，推荐将指令缓存也关闭。

三、寄存器设置

寄存器 `R0` 的值应为 0，`R1` 的值表示机器类型，`R2` 的值则是引导参数列表在内存中的起始地址。这三个寄存器是在最后启动内核时需要设置的。

7.2 使用 U-boot

本节将对 U-boot 界面中使用的命令进行一个汇总，然后通过实例说明其中经常使用的一些命令。

7.2.1 U-boot 主要命令与环境变量

U-boot 提供了许多配置、管理及运行命令，主要命令如表 7.1 所示。

表 7.1 U-boot 基本命令

命令	功能
help	列出 U-boot 所提供的命令列表，或者对某个命令的帮助
bdinfo	显示机器信息
boot	执行环境变量 <code>bootcmd</code> 的内容
bootm	启动 U-boot 映像
cmp	内存比较
cp	内存复制
go	从指定地址执行程序
loadb	通过串口下载文件（ <code>kermit</code> 模式）
md	显示内存值
mm	修改内存值
nand	nand 子系统命令
ping	测试网络连接
printenv	输出环境变量的值
reset	复位
run	执行环境变量的内容
saveenv	保存环境变量到 Flash
setenv	设置环境变量的值
tftp	通过 TFTP 协议下载文件到内存

其中 `nand` 子系统命令其实包含了多个对 NAND Flash 进行操作的命令，如表 7.2 所示。

表 7.2 nand 子系统命令

命令	功能
nand info	列出 NAND Flash 设备信息
nand device	显示或设置当前 NAND Flash 设备
nand read	从当前 NAND Flash 设备读数据到内存
nand write	将内存数据写入当前 NAND Flash 设备
nand erase	对当前 NAND Flash 设备进行擦除操作
nand bad	显示当前 NAND Flash 设备上的坏块
nand read.oob	读当前 NAND Flash 设备的额外数据区
nand write.oob	写当前 NAND Flash 设备的额外数据区

需要注意的是，U-boot 对各种不同的目标机可进行不同的定制，因此所支持的命令集也可能有所差异。

U-boot 有很多环境变量，它们对 U-boot 的使用有很重要的意义，常见的一些环境变量及含义如表 7.3 所示。

表 7.3 U-boot 环境变量

环境变量	含义
stdin	标准输入设备，开发中常设置为串口 serial
stdout	标准输出设备，开发中常设置为串口 serial
stderr	标准错误输出设备，开发中常设置为串口 serial
baudrate	串口波特率，对 HY2410A 开发板应设置为 115200
ethaddr	以太网控制器 MAC 地址
gatewayip	网关地址
netmask	子网掩码
ipaddr	自身 IP 地址
serverip	服务器 IP 地址，与 tftp 等命令的使用有关
bootcmd	U-boot 启动时自动执行的脚本
bootdelay	启动时倒数计时的秒数
bootargs	内核引导参数，根据实际情况进行设置
bootfile	默认的启动映像文件
filesize	所下载文件的大小，下载完成后自动设置
fileaddr	所下载文件的内存地址，下载完成后自动设置

实际上，setenv 命令可以建立新的环境变量，当然这些自定义的环境变量对 U-boot 本身来说没什么意义，但适当的使用可以简化操作，增加 U-boot 环境的可维护性。

这些环境变量还可以在输入命令时引用，如：

```
nand write 30008000 30000 ${filesize}
```

7.2.2 使用实例

下面对几个常用的 U-boot 命令通过实例进行说明。

一、环境变量设置

可用如下命令设置网络下载时 tftp 服务器的地址：

```
setenv serverip 192.168.1.135
```

一般在使用 tftp 网络下载命令前，要设置好 tftp 服务器的 IP 地址。

用如下命令可设置内核的引导参数：

```
setenv bootargs 'console=ttySAC0 root=/dev/mtdblock2 rootfstype=yaffs2  
init=/linuxrc'
```

因为变量值中有空格，所以要用单引号包围起来。其中 console 参数用于设置内核的默认控制台，root 参数用于设置内核根文件系统所在的设备，rootfstype 用于设置根文件系统的类型，init 参数是内核要执行的根文件系统中的初始化程序。

在开发阶段，往往需要使用 NFS 作为根文件系统，引导参数可设置如下：

```
setenv bootargs 'console=ttySAC0 root=nfs nfsroot=192.168.1.129:/home/jyg/work/  
sysroot ip=192.168.1.128 init=/linuxrc'
```

其中 ip 参数是目标机本身的 IP 地址，而 nfsroot 参数则指明了要挂载的根文件系统。

二、读写 NAND Flash

向 Flash 写入数据前需先将相应的块擦除，命令如下：

```
nand erase 0x30000 0x300000
```

这条命令将把 Flash 从地址 0x30000 开始的长度为 0x300000 的区域擦除。注意擦除的区域必须与 Flash 的擦除块边界对齐。

将内存中的数据写入 Flash 的命令如下：

```
nand write 0x30008000 0x30000 0x300000
```

这条命令将把从内存地址 0x30008000 开始的数据写入 Flash 中从地址 0x30000 开始的区域，写入的长度为 0x300000 字节。

将 Flash 中存储的数据读入内存的命令如下：

```
nand read 0x30008000 0x30000 0x300000
```

这条命令将把 Flash 中从地址 0x30000 开始的数据读取到内存中从地址 0x30008000 开始的区域，读取的长度为 0x300000 字节。



一般来说，U-boot 命令中给出的数字默认为十六进制，因此可以省略 0x 前缀。

7.3 U-boot 源码分析

在实际使用中, U-boot 被固化在 CPU 的上电/复位启动地址处(通常在非易失存储器中)。每当嵌入式设备上电/复位时, CPU 总是从启动地址处执行, 也就是执行 U-boot。U-boot 启动后, 首先初始化各种硬件设备, 如 CPU、缓存、存储器、MMU、总线控制器、各种 I/O 接口等, 然后从远程主机或者本地非易失存储设备中装载可执行文件或操作系统, 为整个嵌入式系统准备运行环境。

要使用 U-boot, 最初必须使用某种硬件支持的方式将 U-boot 映像写入非易失存储器中。例如, 在 HY2410A 开发板上, 可以使用 JTAG 接口将 U-boot 映像写入 Flash 的开始处。

本节的主要内容是分析 U-boot 源码的结构, 以便进行 U-boot 的移植。

7.3.1 总体架构与内存布局

U-boot 源码的总体结构如图 7.1 所示。主要部分包括命令行界面、对各种命令的支持、各种硬件驱动以及板级初始化代码。板级初始化代码是系统启动最初执行的代码, 它对各种硬件进行初始化。硬件驱动则提供对硬件进行操作的支持。命令行界面使得用户可以输入各种命令对系统进行控制, 所输入的命令解析后将由各种对命令的支持代码来实现。

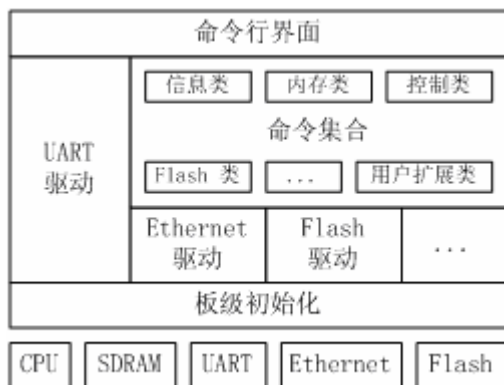


图 7.1 U-boot 总体架构

根据 U-boot 的基本功能, 可以将其代码划分为顺序执行的两大阶段。第一阶段是依赖于 CPU 体系架构的汇编代码, 进行各种初始化准备工作。第二阶段用 C 语言实现, 实现系统的管理、引导功能, 包括操作系统的下载与引导等。

HY2410A 开发板采用 S3C2410A 作为 CPU, 并使用其 NAND Flash 启动方式。由于程序不能在 NAND Flash 中直接运行, 所以在这种启动方式下, CPU 首先将 NAND Flash 的前 4K 字节内容复制到芯片内部位于地址 0 处的一块 SRAM 中, 然后开始执行。因此 U-boot 的第一阶段必须放在这 4K 字节之内, 并且在第一阶段运行结束前将自身的所有代码复制到内存中以便执行。

U-boot 映像进入内存以后放置在较高端的地址, 如图 7.2 所示是在 HY2410A 开发板上它的基本内存布局。

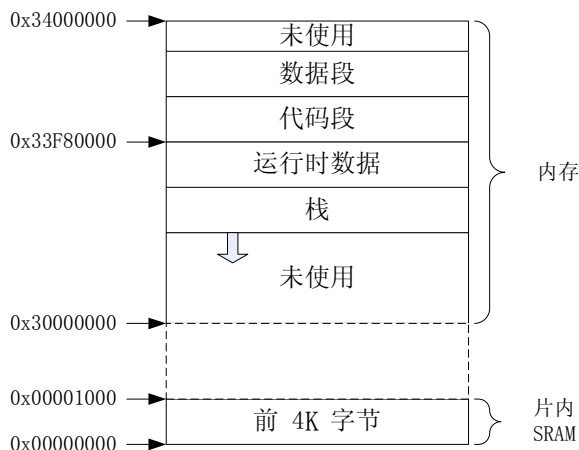


图 7.2 U-boot 内存布局

HY2410A 的内存地址从 0x30000000 开始到 0x34000000。在第一阶段的代码执行完毕后，U-boot 将自身的映像复制到从 0x33F80000 开始的内存区域中，同时还要向下预留一部分内存用于存储全局数据和栈。因此在 U-boot 中执行程序时，要注意下载的程序映像不能将 U-boot 本身所用的空间覆盖。

7.3.2 源码目录

U-boot 采用了一种高度模块化的编程方式，不同功能类别的代码分别放在不同的目录中。下面介绍几个移植 U-boot 时常常用到的目录。

一、board

这个目录存放了所有 U-boot 支持的目标板的子目录。在这些子目录中一般是针对特定目标板的一些初始化和操作代码。要将 U-boot 移植到自己的目标板上，必须参考这个目录下的内容。HY2410A 开发板与 SMDK2410 目标板类似，因此可以在它的基础上进行移植，也就是说，我们只需要关心 smdk2410 目录的内容。

二、cpu

这个目录存放了 U-boot 支持的 CPU 类型，一般是针对特定 CPU 的初始化和操作代码，因为 S3C2410A 是基于 ARM920T 核的 CPU，所以移植时只需要关心其中的 arm920t 目录。目录中的 start.S 是 U-boot 的第一阶段代码，它的主要工作就是对整个 U-boot 目标代码的重定位，即将 U-boot 复制到内存中去运行。

三、common

这个目录存放独立于处理器体系架构的通用代码，包括 U-boot 的一些公共命令的实现，如内存大小探测与故障检测等。一般来说，其中以 cmd_*.c 命名的文件就是对相应命令的实现，如 cmd_bootm.c 就是对命令 bootm 的实现。

四、drivers

这个目录存放的是各种外设接口的驱动程序，如以太网驱动等。



五、fs

这个目录中存放了 U-boot 支持的文件系统。

六、include

这个目录是存放各种 CPU 及目标板的头文件和配置文件的公共目录,其中的 configs 目录中存放了各种目标板的配置头文件,如 smdk2410.h 中定义了 SMDK2410 开发板相关的配置参数,这些参数在移植时应根据实际情况进行修改。

七、lib_XXX

用于存放 XXX 体系架构的处理器的相关支持。如 lib_arm 目录就包含 ARM 平台的公共接口代码文件。

八、net

这个目录用于存放与网络功能相关的文件,如 bootp, nfs, tftp 等。

7.3.3 实现分析

下面对 U-boot 源码中几个关键功能的实现进行分析,主要包括以下几个方面。

- ◆ 使用汇编语言编写的第一阶段代码。
- ◆ 第二阶段代码命令的实现。
- ◆ 第二阶段操作系统引导机制的实现。

代码来自 U-boot 1.2.0 源码。其中有些代码是移植到 HY2410A 开发板以后新增的,并非初始的源码所有。摘录代码时去除了与 HY2410A 开发板无关的代码分支并重新进行了注释。

7.3.3.1 第一阶段代码

第一阶段的代码主要包括以下执行步骤。

- ◆ 一些基本的硬件初始化工作。
- ◆ 将 U-boot 映像复制到 RAM 空间。
- ◆ 跳转到第二阶段代码的入口点。

以 HY2410A 开发板为例,对应的第一阶段代码放在 cpu/arm920t/start.S 文件中。其入口处的代码如下:

```
.globl _start
_start: b      reset
        ldr     pc, _undefined_instruction
        ldr     pc, _software_interrupt
        ldr     pc, _prefetch_abort
        ldr     pc, _data_abort
        ldr     pc, _not_used
        ldr     pc, _irq
        ldr     pc, _fiq
```



```

_undefined_instruction: .word undefined_instruction
_software_interrupt:   .word software_interrupt
_prefetch_abort:      .word prefetch_abort
_data_abort:          .word data_abort
_not_used:             .word not_used
_irq:                  .word irq
_fiq:                  .word fiq

```

实际上这就是异常向量表。当系统上电或复位时，将执行第一条指令，即跳转到标签为 `reset` 的代码处，摘录如下：

```

reset:
    /* 将 CPU 设为 SVC32 模式 */
    mrs    r0,cpsr
    bic    r0,r0,#0x1f
    orr    r0,r0,#0xd3
    msr    cpsr,r0

    /* 关闭看门狗 */
    # define pWTCN      0x53000000    /* 看门狗寄存器 */
    # define INTMSK      0x4A000008    /* 中断屏蔽寄存器 */
    # define INTSUBMSK    0x4A00001C    /* 次级中断屏蔽寄存器 */
    # define CLKDIVN      0x4C000014    /* 时钟分频寄存器 */

    ldr    r0, =pWTCN
    mov    r1, #0x0
    str    r1, [r0]

    /* 屏蔽所有中断 */
    mov    r1, #0xffffffff
    ldr    r0, =INTMSK
    str    r1, [r0]
    ldr    r1, =0x7ff
    ldr    r0, =INTSUBMSK
    str    r1, [r0]

    /* 设置时钟 */
    ldr    r0, =CLKDIVN
    mov    r1, #3
    str    r1, [r0]

    bl     cpu_init_crit

```

这段代码主要是将 CPU 设为管理模式，关闭看门狗，屏蔽中断并设置时钟，最后调用 `cpu_init_crit` 函数进行 CPU 的初始化，函数的代码如下：

```

cpu_init_crit:
    /* 清除指令和数据缓存 */
    mov    r0, #0
    mcr    p15, 0, r0, c7, c7, 0    /* flush v3/v4 cache */

```



```

mcr    p15, 0, r0, c8, c7, 0    /* flush v4 TLB */

/* 禁用 MMU 和缓存 */
mrc    p15, 0, r0, c1, c0, 0
bic    r0, r0, #0x00002300      @ clear bits 13, 9:8 (--V- --RS)
bic    r0, r0, #0x00000087      @ clear bits 7, 2:0 (B--- -CAM)
orr    r0, r0, #0x00000002      @ set bit 2 (A) Align
orr    r0, r0, #0x00001000      @ set bit 12 (I) I-Cache
mcr    p15, 0, r0, c1, c0, 0

/* 设置 SDRAM 控制器, 与具体的目标板相关 */
mov    ip, lr
bl     lowlevel_init
mov    lr, ip
mov    pc, lr

```

在这个函数中主要做了以下操作：清除指令与数据缓存，禁用 MMU 与数据指令缓存，最后调用 `lowlevel_init` 函数设置 SDRAM 控制器。`lowlevel_init` 函数的实现是与具体的目标板有关的，在 `board/smdk2410/lowlevel_init.S` 文件中。

现在回到 `reset` 标签后的代码，从函数 `cpu_init_crit` 返回后的代码如下：

```

copy_myself:
    @ 复位 NAND
    mov    r1, #0x4E000000        @ NAND Flash 控制器基地址
    ldr    r2, =0xf830            @ NFCONF 寄存器的值
    str    r2, [r1, #oNFCONF]
    ldr    r2, [r1, #oNFCONF]
    bic    r2, r2, #0x800        @ 使能 Flash 片选
    str    r2, [r1, #oNFCONF]
    mov    r2, #0xff            @ 复位命令
    strb   r2, [r1, #oNFCMD]
    mov    r3, #0                @ 延时
1:    add    r3, r3, #0x1
    cmp    r3, #0xa
    blt    1b
2:    ldr    r2, [r1, #oNFSTAT]    @ 读 Flash 状态直至其就绪
    tst    r2, #0x1
    beq    2b
    ldr    r2, [r1, #oNFCONF]
    orr    r2, r2, #0x800        @ 禁用 Flash 片选
    str    r2, [r1, #oNFCONF]

    /* 设置堆栈 */
stack_setup:
    ldr    r0, _TEXT_BASE        /* 预留 U-boot 本身所占空间 */
    sub    r0, r0, #CFG_MALLOC_LEN /* 预留动态数据空间 */
    sub    r0, r0, #CFG_GBL_DATA_SIZE /* 预留板信息所占空间 */
    sub    sp, r0, #12          /* 预留 12 个字节作为 abort 栈 */

    @ 将 U-boot 本身复制到内存

```

```

        ldr    r0, _TEXT_BASE
        mov    r1, #0x0
        mov    r2, #CFG_UBOOT_SIZE
        bl     nand_read_ll

        tst    r0, #0x0
        beq    ok_nand_read

ok_nand_read:
    @ verify
    mov    r0, #0
    ldr    r1, _TEXT_BASE
    mov    r2, #0x400        @ 4 bytes * 1024 = 4K-bytes
go_next:
    ldr    r3, [r0], #4
    ldr    r4, [r1], #4
    teq    r3, r4
    bne    notmatch
    subs   r2, r2, #4
    beq    done_nand_read
    bne    go_next
notmatch:
1:      b     1b
done_nand_read:

clear_bss:
    ldr    r0, _bss_start        /* bss 段的开始 */
    ldr    r1, _bss_end          /* bss 段结束 */
    mov    r2, #0x00000000       /* 准备清空 */

clbss_1: str    r2, [r0]          /* 用循环清空 bss 段 */
        add    r0, r0, #4
        cmp    r0, r1
        ble    clbss_1

        ldr    pc, _start_armboot
_start_armboot: .word start_armboot

```

这段代码主要执行了以下任务：设置堆栈指针，将 U-boot 映像从 NAND Flash 复制到内存，以及清空 bbs 数据段内容。复制操作主要利用了 `nand_read_ll` 函数，这是一个 C 语言函数，故在调用它之前应先设置好堆栈指针。注意 SMDK2410 开发板本身并不支持 NAND Flash，因此这些 NAND Flash 相关代码都是移植到 HY2410A 之后增加的。

代码的最后是将 `start_armboot` 这个地址值放入 PC 寄存器，这样就完成了第一阶段的所有工作。`start_armboot` 是一个 C 语言函数，定义在文件 `lib_arm/board.c` 中，它是第二阶段代码的入口。

在第二阶段代码中，将进行更多的初始化工作，如对各种设备和接口的初始化、串口终端的初始化等。如果没有设置自动运行，则最终将进入一个循环，在循环内读取用户输入的命令并执行。

7.3.3.2 U-boot 命令实现

U-boot 的软件架构是高度可扩展的，其中很重要的一个方面就体现在其可扩展的命令结构上。

下面以 go 命令为例讨论 U-boot 命令结构的实现。

go 命令实现在 common/cmd_boot.c 文件中，在这个文件中有如下的代码：

```
U_BOOT_CMD(
    go, CFG_MAXARGS, 1,    do_go,
    "go      - start application at address 'addr'\n",
    "addr [arg ...]\n    - start application at address 'addr'\n"
    "          passing 'arg' as arguments\n"
);
```

U-boot 中的所有命令都对应着这样一段代码。U_BOOT_CMD 是一个宏，它将展开成一个关于命令的结构体数据，特别是其中包含了实现这个命令的函数入口地址，如上述代码说明 do_go 函数是 go 命令的入口地址。

U_BOOT_CMD 宏定义在文件 include/command.h 中，定义的代码如下：

```
#define Struct_Section __attribute__((unused,section (".u_boot_cmd")))

#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, \
    usage, help}
```

可见 U_BOOT_CMD 宏等价于定义了一个 cmd_tbl_t 型的变量，并且这个变量放在一个特殊的段 .u_boot_cmd 中。cmd_tbl_t 类型的定义如下：

```
struct cmd_tbl_s {
    char *name;          /* 命令名称 */
    int maxargs;         /* 最大参数个数 */
    int repeatable;      /* 是否允许自动重复 */
    int (*cmd)(struct cmd_tbl_s *, int, int, char *[]); /* 实现函数 */
    char *usage;         /* 帮助信息（短） */
    char *help;          /* 帮助信息（长） */
};

typedef struct cmd_tbl_s      cmd_tbl_t;
```

也就是说，我们使用下面的代码就可以定义一个命令的数据结构：

```
U_BOOT_CMD(name, maxargs, rep, cmd, usage, help);
```

其各个参数的含义解释如下。

- ◆ name：命令的名称。
- ◆ maxargs：最大参数个数，注意包括命令名本身。
- ◆ rep：指示命令是否可重复运行，也就是在 U-boot 命令环境下，如果直接按回车键，系统

会自动重复执行上一次的命令。

- ◆ cmd: 命令实现的函数指针。
- ◆ usage: 使用命令的提示信息。
- ◆ help: 帮助信息。

对照宏的定义, go 命令的定义展开如下:

```
cmd_tbl_t __u_boot_cmd_go __attribute__((unused,section (".u_boot_cmd")))= {
    "go",
    CFG_MAXARG,
    1,
    do_go,
    "addr [arg ...]\n    - start application at address 'addr',\n    'passing 'arg' as arguments"
};
```

__attribute__ 指定 gcc 编译属性, 指示将这个结构体数据放在 .u_boot_cmd 区段。在 U-boot 的链接脚本 board/smdk2410/u-boot.lds 中有如下定义:

```
__u_boot_cmd_start = .;
.u_boot_cmd : { *(.u_boot_cmd) }
__u_boot_cmd_end = .;
```

这段代码将 .u_boot_cmd 的起始地址定义为 __u_boot_cmd_start, 而结束地址定义为 __u_boot_cmd_end, 在程序中可以将它们作为外部全局变量使用, 因此能够遍历所有命令的数据、通过命令的名称查找到要执行的函数及帮助信息等。

7.3.3.3 引导内核的实现

ARM Linux 内核对 bootloader 的引导功能有一定的要求, 在执行内核代码前必须设置下列条件。

- ◆ 对 CPU 寄存器的设置为 R0 = 0, R1 = 机器类型 ID, R2 = 引导参数列表的地址。
- ◆ 必须禁止中断 (IRQ 与 FIQ)。
- ◆ CPU 必须处于 SVC 模式。
- ◆ MMU 必须关闭。
- ◆ 数据缓存必须关闭。

以上条件中最重要的一点是设置内核的引导参数。目前 Linux 支持两种格式的引导参数。

一、旧的内核引导参数结构

旧的引导参数使用一个结构体传递信息。在 U-boot 源码中, 这个结构体定义在文件 include/asm-arm/setup.h 中, 代码如下:

```
#define COMMAND_LINE_SIZE 1024

struct param_struct {
```

```

union {
    struct {
        unsigned long page_size;           /* 0 */
        unsigned long nr_pages;            /* 4 */
        unsigned long ramdisk_size;        /* 8 */
        unsigned long flags;               /* 12 */
#define FLAG_READONLY    1
#define FLAG_RDLOAD      4
#define FLAG_RDPROMPT    8
        unsigned long rootdev;             /* 16 */
        unsigned long video_num_cols;      /* 20 */
        unsigned long video_num_rows;      /* 24 */
        unsigned long video_x;             /* 28 */
        unsigned long video_y;             /* 32 */
        unsigned long memc_control_reg;     /* 36 */
        unsigned char sounddefault;         /* 40 */
        unsigned char adfsdrives;          /* 41 */
        unsigned char bytes_per_char_h;     /* 42 */
        unsigned char bytes_per_char_v;     /* 43 */
        unsigned long pages_in_bank[4];     /* 44 */
        unsigned long pages_in_vram;        /* 60 */
        unsigned long initrd_start;         /* 64 */
        unsigned long initrd_size;          /* 68 */
        unsigned long rd_start;             /* 72 */
        unsigned long system_rev;           /* 76 */
        unsigned long system_serial_low;    /* 80 */
        unsigned long system_serial_high;   /* 84 */
        unsigned long mem_fclk_21285;       /* 88 */
    } s;
    char unused[256];
    } u1;
    union {
        char paths[8][128];
        struct {
            unsigned long magic;
            char n[1024 - sizeof(unsigned long)];
        } s;
    } u2;
    char commandline[COMMAND_LINE_SIZE];
};

```

上述结构体共占用内存为 $(256 + 1024 + 1024)$ 字节。其中第一个联合体占用内存为 256 字节，包含的重要内容为内存相关的信息，如内存页的大小 `page_size` 与内存页的个数 `nr_pages` 等。`commandline` 成员为 U-boot 要传递给内核的引导参数，即由 `bootargs` 环境变量指定的参数。

二、标签列表

Linux 内核的维护者希望采用新的方式来传递引导参数，这就是所谓的标签列表。这种方式比之前采用结构体数据的方式更灵活，对参数的描述更细致。

标签列表中的每个标签由标签头和标签体组成。标签头说明这个标签的大小（注意单位是整数，

而不是字节)以及这个标签的类型。标签的类型是由内核定义好的一个数字。标签头用一个结构体 `struct tag_header` 表示,其定义如下:

```
struct tag_header {
    u32 size;
    u32 tag;
};
```

在标签头之后,根据标签的类型,所需的标签体也不同。下面看一个例子, `ATAG_CORE` 是内核要求放在引导参数中的第一个标签,它的定义如下:

```
#define ATAG_CORE      0x54410001
struct tag_core {
    u32 flags;
    u32 pagesize;
    u32 rootdev;
};
```

上述 `ATAG_CORE` 就是这个标签的类型,必须与内核的定义一致,而 `struct tag_core` 类型定义的就是它的标签体。

标签列表的结束由一个特殊标签类型 `ATAG_NONE` 标志,它没有标签体。

下面列举两个重要的标签类型。

一、ATAG_MEM

它用来设置内存信息,标签体定义如下:

```
struct tag_mem32 {
    u32    size; /* 内存大小 */
    u32    start; /* 物理内存起始地址 */
};
```

二、ATAG_CMDLINE

它用来传递命令行参数,即 U-boot 的 `bootargs` 变量的内容,其标签体定义如下:

```
struct tag_cmdline {
    char    cmdline[1];
};
```

注意这里只定义了一个字符的空间,但实际使用时长度是可变的。

U-boot 源码中给出了一些设置标签列表的代码,放在文件 `lib_arm/armlinux.c` 中。设置的方法是这样的,首先定义一个全局变量:

```
static struct tag *params;
```

其中 `struct tag` 类型是一个将所有标签类型组合在一起的结构体,定义如下:

```
struct tag {
    struct tag_header hdr;
    union {
```

```

    struct tag_core      core;
    struct tag_mem32      mem;
    struct tag_videotext  videotext;
    struct tag_ramdisk    ramdisk;
    struct tag_initrd     initrd;
    struct tag_serialnr   serialnr;
    struct tag_revision   revision;
    struct tag_videolfb    videolfb;
    struct tag_cmdline    cmdline;
    struct tag_acorn       acorn;
    struct tag_memclk      memclk;
} u;
};

```

所有标签的头格式都是相同的，只是标签体不同，因此用联合体的方式将它们组合在一起。下面是设置起始标签的函数代码：

```

static void setup_start_tag (bd_t *bd)
{
    params = (struct tag *) bd->bi_boot_params;
    params->hdr.tag = ATAG_CORE;
    params->hdr.size = tag_size (tag_core);
    params->u.core.flags = 0;
    params->u.core.pagesize = 0;
    params->u.core.rootdev = 0;
    params = tag_next (params);
}

```

在这个函数内，首先将变量 `params` 的值设为标签列表的开始地址，然后逐个设置标签中的成员，最后 `params` 变量的值将指向下一个标签应该放置的地址。

`tag_size` 是一个宏，用于得到标签的大小，定义如下：

```
#define tag_size(type) ((sizeof(struct tag_header) + sizeof(struct type)) >> 2)
```

可见标签的大小是标签头与标签体的大小之和。`tag_next` 也是一个宏，定义如下：

```
#define tag_next(t) ((struct tag *)((u32 *) (t) + (t)->hdr.size))
```

可见标签在内存中是连续存放的。最后，用于设置结束标签的函数如下：

```

static void setup_end_tag (bd_t *bd)
{
    params->hdr.tag = ATAG_NONE;
    params->hdr.size = 0;
}

```

由于旧的参数传递方式比较简单，所以在 HY2410A 的 U-boot 中仍然采用了旧的方式，用于设置参数列表的代码如下：

```
{
```

```

struct param_struct *params = (struct param_struct *)bd->bi_boot_params;
memset(params, 0, sizeof(struct param_struct));
params->ul.s.page_size = 0x00001000; /* 设置内存页大小 */
params->ul.s.nr_pages = (0x04000000 >> 12); /* 设置内存页数 */
strcpy(params->commandline, commandline); /* 复制命令行参数 */
}

```

这段代码主要是将 `commandline` 字符串的内容复制到了参数列表中,这个字符串是在前面从 `bootargs` 参数中得到的,代码如下:

```
char *commandline = getenv ("bootargs");
```

当输入 `bootm` 命令引导内核时,首先执行的是文件 `common/cmd_bootm.c` 中的 `do_bootm` 函数,函数中将根据内核 U-boot 映像的头部数据判断操作系统的类型,如果是 Linux,则最终调用 `do_bootm_linux` 函数,它在文件 `lib_arm/armlinux.c` 中,其中就包含了上述的设置引导参数的代码。

下面说明 `do_bootm_linux` 函数中引导内核的方法。首先获得内核映像的入口地址,代码如下:

```
theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep);
```

这里 `hdr` 是指向内核 U-boot 映像头部数据的指针,而 `hdr->ih_ep` 就是内核的入口地址。最后用下述代码调用内核:

```
theKernel (0, bd->bi_arch_number, bd->bi_boot_params);
```

根据 ATPCS 调用约定,上述函数的三个参数分别放在寄存器 R0, R1, R2 中,于是实现了内核要求的入口条件。

7.4 U-boot 移植

移植 U-boot 到一款主流的处理器并不需要改动很多代码,因为整个 U-boot 的源码是高度模块化的,层次结构分明,一般来说,只需要将特定的硬件部分修改为对应的处理器支持即可。特别是 U-boot 中对 ARM 处理器的支持已经比较完善,如果要移植到 ARM 处理器的目标板,我们只需要找到源码中已有的一款类似的目标板,根据自己的硬件配置情况进行修改即可。

一般来说,嵌入式处理器都有丰富的外部接口资源,但对于 `bootloader` 来说并不需要支持很多接口,只要能够引导内核即可。因此在移植 U-boot 时,重点是那些必须的功能接口,如串口、以太网接口等。当 U-boot 移植成功能够运行起来后,可以再根据需要添加更多的功能。

本节将以 HY2410A 开发板为例介绍如何将 U-boot 移植到自己的开发板上。

7.4.1 源码修改

首先取得一个 U-boot 源码包,可用以下命令:

```
wget ftp://ftp.denx.de/pub/u-boot/u-boot-1.2.0.tar.bz2
```

这样下载的是 U-boot 1.2.0 的源码，将其解压缩后出现 u-boot-1.2.0 目录，这就是我们的工作目录。

由于 HY2410A 开发板与源码中提供的 SMDK2410 开发板类似，所以可以在它的基础上进行移植。下面将以 diff 文件的形式说明对源码的修改。diff 文件的格式简单说明如下。

- ◆ 第一列是减号的行表示从源码中删掉的行。
- ◆ 第一列是加号的行表示要增加的行。
- ◆ 第一列是空格的行只是为了帮助定位，不改动。
- ◆ 在每一段修改前都有一行指示修改的位置和行数。

对源码改动最多的地方是因为原来的 SMDK2410 只支持 NOR Flash，而 HY2410A 支持的是 NAND Flash。下面分别对每个修改的文件进行说明。

一、board/smdk2410/Makefile

```
@@ -25,7 +25,7 @@

LIB      = $(obj)lib$(BOARD).a

-COBSJS  := smdk2410.o flash.o
+COBSJS  := smdk2410.o nand_read.o
SOBSJS   := lowlevel_init.o

SRCS     := $(SOBSJS:.o=.S) $(COBSJS:.o=.c)
```

对这个 Makefile 修改的原因是我们增加了一个源码 nand_read.c，去掉了一个源码 flash.c。

二、board/smdk2410/nand_read.c

```
@@ -0,0 +1,77 @@
+/*
+ * vivi/s3c2410/nand_read.c: Simple NAND read functions for booting from NAND
+ *
+ * This is used by cpu/arm920/start.S assembler code,
+ * and the board-specific linker script must make sure this
+ * file is linked within the first 4kB of NAND flash.
+ *
+ * Taken from GPLv2 licensed vivi bootloader,
+ * Copyright (C) 2002 MIZI Research, Inc.
+ *
+ * Author: Hwang, Chideok <hwang@mizi.com>
+ * Date  : $Date: 2002/08/14 10:26:47 $
+ */
+
+#include <config.h>
+
+#ifdef CONFIG_S3C2410_NAND_BOOT
+
+#define __REGb(x)    (*(volatile unsigned char *)(x))
```

```

#define __REGi(x)    (*(volatile unsigned int *)(x))
#define NF_BASE      0x4e000000
#define NFCONF        __REGi(NF_BASE + 0x0)
#define NFCMD         __REGb(NF_BASE + 0x4)
#define NFADDR        __REGb(NF_BASE + 0x8)
#define NFDATA        __REGb(NF_BASE + 0xc)
#define NFSTAT        __REGb(NF_BASE + 0x10)
+
#define BUSY 1
+inline void wait_idle(void) {
+    int i;
+
+    while(!(NFSTAT & BUSY))
+        for(i=0; i<10; i++);
+}
+
#define NAND_SECTOR_SIZE    512
#define NAND_BLOCK_MASK    (NAND_SECTOR_SIZE - 1)
+
/* low level nand read function */
+int
+nand_read_ll(unsigned char *buf, unsigned long start_addr, int size)
+{
+    int i, j;
+
+    if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK)) {
+        return -1;    /* invalid alignment */
+    }
+
+    /* chip Enable */
+    NFCONF &= ~0x800;
+    for(i=0; i<10; i++);
+
+    for(i=start_addr; i < (start_addr + size);) {
+        /* READ0 */
+        NFCMD = 0;
+
+        /* Write Address */
+        NFADDR = i & 0xff;
+        NFADDR = (i >> 9) & 0xff;
+        NFADDR = (i >> 17) & 0xff;
+        NFADDR = (i >> 25) & 0xff;
+
+        wait_idle();
+
+        for(j=0; j < NAND_SECTOR_SIZE; j++, i++) {
+            *buf = (NFDATA & 0xff);
+            buf++;
+        }
+    }
+}
+

```



```
+ /* chip Disable */
+ NFCCONF |= 0x800; /* chip disable */
+
+ return 0;
+}
+
+#endif /* CONFIG_S3C2410_NAND_BOOT */
```

这是 `nand_read.c` 的主要内容，它主要实现了一个函数 `nand_read_ll`，用来从 NAND Flash 读 U-boot 映像到内存。

三、board/smdk2410/smdk2410.c

```

@@ -36,10 +36,10 @@
#define M_MDIV    0xC3
#define M_PDIV    0x4
#define M_SDIV    0x1
-#elif FCLK_SPEED==1          /* Fout = 202.8MHz */
-#define M_MDIV    0xA1
-#define M_PDIV    0x3
-#define M_SDIV    0x1
+#elif FCLK_SPEED==1          /* Fout = 200MHz */
+#define M_MDIV    0x5C
+#define M_PDIV    0x4
+#define M_SDIV    0x0
#endif

#define USB_CLOCK 1
@@ -87,7 +87,7 @@

/* set up the I/O ports */
gpio->GPACON = 0x007FFFFFFF;
- gpio->GPBCON = 0x00044555;
+ gpio->GPBCON = 0x00155555;
gpio->GPBUP  = 0x000007FF;
gpio->GPCCON = 0xAAAAAAAA;
gpio->GPCUP  = 0x0000FFFF;
@@ -99,11 +99,11 @@
gpio->GPFUP  = 0x000000FF;
gpio->GPGCON = 0xFF95FFBA;
gpio->GPGUP  = 0x0000FFFF;
- gpio->GPHCON = 0x002AFAAA;
+ gpio->GPHCON = 0x0016FAAA;
gpio->GPHUP  = 0x000007FF;

/* arch number of SMDK2410-Board */
- gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;
+ gd->bd->bi_arch_number = 2247; /* MACH_HY2410 */

/* adress of boot parameters */
gd->bd->bi_boot_params = 0x30000100;

```



这里主要修改了 SMDK2410 板级初始化的几个参数，如时钟的分频、GPIO 管脚的配置以及机器类型 ID 的设置。

四、common/env_nand.c

```
@@ -57,8 +57,15 @@
    size_t start, size_t len,
    size_t * retlen, u_char * buf);

#ifdef CFG_NAND_LEGACY
+extern struct nand_chip nand_dev_desc[CFG_MAX_NAND_DEVICE];
+extern int nand_legacy_erase(struct nand_chip *nand, size_t ofs,
+    size_t len, int clean);
+nand_info_t nand_info[CFG_MAX_NAND_DEVICE];
#else
/* info for NAND chips, defined in drivers/nand/nand.c */
extern nand_info_t nand_info[];
#endif

/* references to names in env_common.c */
extern uchar default_environment[];
@@ -76,9 +83,7 @@

/* local functions */
-#if !defined(ENV_IS_EMBEDDED)
    static void use_default(void);
-#endif

DECLARE_GLOBAL_DATA_PTR;

@@ -192,12 +197,22 @@
    int ret = 0;

    puts ("Erasing Nand...");
+if (CFG_NAND_LEGACY)
+    if (nand_legacy_erase(nand_dev_desc+0, CFG_ENV_OFFSET, CFG_ENV_SIZE, 0))
+        return 1;
+else
+    if (nand_erase(&nand_info[0], CFG_ENV_OFFSET, CFG_ENV_SIZE))
+        return 1;
+endif

    puts ("Writing to Nand... ");
    total = CFG_ENV_SIZE;
+if (CFG_NAND_LEGACY)
+    ret = nand_legacy_rw(nand_dev_desc+0, 0x00 | 0x02, CFG_ENV_OFFSET,
+        total, &total, (u_char*)env_ptr);
+else
    ret = nand_write(&nand_info[0], CFG_ENV_OFFSET, &total, (u_char*)env_ptr);
```



```

+endif
    if (ret || total != CFG_ENV_SIZE)
        return 1;

@@ -272,7 +287,12 @@
    int ret;

    total = CFG_ENV_SIZE;
+if (CFG_NAND_LEGACY)
+    ret = nand_legacy_rw(nand_dev_desc+0, 0x01 | 0x02, CFG_ENV_OFFSET,
+        total, &total, (u_char*)env_ptr);
+else
    ret = nand_read(&nand_info[0], CFG_ENV_OFFSET, &total, (u_char*)env_ptr);
+endif
    if (ret || total != CFG_ENV_SIZE)
        return use_default();

@@ -282,7 +302,6 @@
    }
    #endif /* CFG_ENV_OFFSET_REDUND */

-#if !defined(ENV_IS_EMBEDDED)
    static void use_default()
    {
        puts ("*** Warning - bad CRC or NAND, using default environment\n\n");
@@ -300,6 +319,5 @@
        gd->env_valid = 1;

    }
-#endif

    #endif /* CFG_ENV_IS_IN_NAND */

```

这里主要修改的是从 Flash 读取环境变量的方法。

五、cpu/arm920t/start.S

```

@@ -140,7 +140,7 @@
    ldr    r0, =INTMSK
    str    r1, [r0]
    # if defined(CONFIG_S3C2410)
-    ldr    r1, =0x3ff
+    ldr    r1, =0x7ff /* not necessary */
    ldr    r0, =INTSUBMSK
    str    r1, [r0]
    # endif
@@ -179,6 +179,29 @@
    ble    copy_loop
    #endif /* CONFIG_SKIP_RELOCATE_UBOOT */

+ifdef CONFIG_S3C2410_NAND_BOOT

```



```

+copy_myself:
+   @ reset NAND
+   mov    r1, #0x4E000000      @ NAND control base
+   ldr     r2, =0xf830         @ initial value enable
+   str     r2, [r1, #oNFCONF]
+   ldr     r2, [r1, #oNFCONF]
+   bic     r2, r2, #0x800      @ enable chip
+   str     r2, [r1, #oNFCONF]
+   mov     r2, #0xff           @ RESET command
+   strb    r2, [r1, #oNFCMD]
+   mov     r3, #0              @ wait
+1:   add    r3, r3, #0x1
+   cmp     r3, #0xa
+   blt     1b
+2:   ldr     r2, [r1, #oNFSTAT] @ wait ready
+   tst     r2, #0x1
+   beq     2b
+   ldr     r2, [r1, #oNFCONF]
+   orr     r2, r2, #0x800      @ disable chip
+   str     r2, [r1, #oNFCONF]
+#endif
+
+   /* Set up the stack */
+   stack_setup:
+       ldr     r0, _TEXT_BASE    /* upper 128 KiB: relocated uboot */
+@@ -189,6 +212,34 @@
+   #endif
+       sub     sp, r0, #12       /* leave 3 words for abort-stack */
+
+#ifdef CONFIG_S3C2410_NAND_BOOT
+   @ copy u-boot to RAM
+   ldr     r0, _TEXT_BASE
+   mov     r1, #0x0
+   mov     r2, #CFG_UBOOT_SIZE
+   bl      nand_read_ll
+
+   tst     r0, #0x0
+   beq     ok_nand_read
+
+ok_nand_read:
+   @ verify
+   mov     r0, #0
+   ldr     r1, _TEXT_BASE
+   mov     r2, #0x400          @ 4 bytes * 1024 = 4K-bytes
+go_next:
+   ldr     r3, [r0], #4
+   ldr     r4, [r1], #4
+   teq     r3, r4
+   bne     notmatch
+   subs    r2, r2, #4
+   beq     done_nand_read

```



```

+   bne    go_next
+notmatch:
+1:   b     1b
+done_nand_read:
+#endif
+
+   clear_bss:
+       ldr    r0, _bss_start      /* find start of bss segment */
+       ldr    r1, _bss_end       /* stop here */

```

这里主要增加了从 NAND Flash 读 U-boot 映像到内存的代码。

六、drivers/nand_legacy/nand_legacy.c

```

@@ -14,6 +14,7 @@
#include <malloc.h>
#include <asm/io.h>
#include <watchdog.h>
+#include <s3c2410.h>

#ifdef CONFIG_SHOW_BOOT_PROGRESS
# include <status_led.h>
@@ -61,6 +62,144 @@
#define NANDRW_JFFS2      0x02
#define NANDRW_JFFS2_SKIP 0x04

+/*
+ * NAND flash initialization.
+ */
+
+typedef enum {
+    NFCE_LOW,
+    NFCE_HIGH
+} NFCE_STATE;
+
+static inline void NF_Reset(void);
+static inline void NF_Init(void);
+static void NF_Conf(u16 conf);
+
+static void NF_Cmd(u8 cmd);
+static void NF_CmdW(u8 cmd);
+static void NF_Addr(u8 addr);
+static void NF_SetCE(NFCE_STATE s);
+static void NF_WaitRB(void);
+static void NF_Write(u8 data);
+static u8 NF_Read(void);
+static void NF_Init_ECC(void);
+static u32 NF_Read_ECC(void);
+
+static inline void NF_Reset(void)
+{

```

```

+   int i;
+
+   NF_SetCE(NFCE_LOW);
+   NF_Cmd(0xFF);           // reset command
+   for(i = 0; i < 10; i++); // tWB = 100ns.
+   NF_WaitRB();            // wait 200~500us;
+   NF_SetCE(NFCE_HIGH);
+}
+
+void nand_init(void)
+{
+   S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
+
+   NF_Init();
+
+   printf ("%4lu MB\n", nand_probe((ulong)nand) >> 20);
+}
+
+static inline void NF_Init(void)
+{
+   #if 0           // a little bit too optimistic
+   +   #define TACLS 0
+   +   #define TWRPH0 3
+   +   #define TWRPH1 0
+   #else
+   +   #define TACLS 0
+   +   #define TWRPH0 4
+   +   #define TWRPH1 2
+   #endif
+
+   NF_Conf((1<<15)|(0<<14)|(0<<13)|(1<<12)|(1<<11)|
+           (TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0));
+   // 1 1 1 1, 1 xxx, r xxx, r xxx
+   // En 512B 4step ECCR nFCE=H tACLS tWRPH0 tWRPH1
+
+   NF_Reset();
+}
+
+static void NF_Conf(u16 conf)
+{
+   S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
+
+   nand->NFCONF = conf;
+}
+
+static void NF_Cmd(u8 cmd)
+{
+   S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
+
+   nand->NFCMD = cmd;
+}

```

```

+
+static void NF_CmdW(u8 cmd)
+{
+    NF_Cmd(cmd);
+    udelay(1);
+}
+
+static void NF_Addr(u8 addr)
+{
+    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
+
+    nand->NFADDR = addr;
+}
+
+static void NF_SetCE(NFCE_STATE s)
+{
+    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
+
+    switch (s) {
+        case NFCE_LOW:
+            nand->NFCONF &= ~(1<<11);
+            break;
+        case NFCE_HIGH:
+            nand->NFCONF |= (1<<11);
+            break;
+    }
+}
+
+static void NF_WaitRB(void)
+{
+    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
+
+    while (!(nand->NFSTAT & (1<<0)));
+}
+
+static void NF_Write(u8 data)
+{
+    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
+
+    nand->NFDATA = data;
+}
+
+static u8 NF_Read(void)
+{
+    S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
+
+    return(nand->NFDATA);
+}
+
+static void NF_Init_ECC(void)
+{

```



```
+   S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
+
+   nand->NFCONF |= (1<<12);
+}
+
+static u32 NF_Read_ECC(void)
+{
+   S3C2410_NAND * const nand = S3C2410_GetBase_NAND();
+
+   return(nand->NFECC);
+}

/*
 * Exported variables etc.
```

这里主要增加了一些对 NAND Flash 进行操作的基本函数。

七、include/configs/smdk2410.h

```
@@ -78,25 +78,28 @@
#define CONFIG_COMMANDS \
        (CONFIG_CMD_DFL      | \
         CFG_CMD_CACHE      | \
-        /*CFG_CMD_NAND     */ \
+        CFG_CMD_NAND      | \
+        CFG_CMD_ENV       | \
+        /*CFG_CMD_EEPROM   */ \
+        /*CFG_CMD_I2C      */ \
+        /*CFG_CMD_USB      */ \
+        CFG_CMD_NET       | \
+        CFG_CMD_PING      | \
+        CFG_CMD_REGINFO   | \
+        CFG_CMD_DATE      | \
-        CFG_CMD_ELF)
+        CFG_CMD_ELF) & ~CFG_CMD_IMLS & ~CFG_CMD_FLASH

/* this must be included AFTER the definition of CONFIG_COMMANDS (if any) */
#include <cmd_confdefs.h>

#define CONFIG_BOOTDELAY      3
-/* #define CONFIG_BOOTARGS      "root=ramfs devfs=mount console=ttySA0,9600" */
-/* #define CONFIG_ETHADDR      08:00:3e:26:0a:5b */
-#define CONFIG_NETMASK      255.255.255.0
-#define CONFIG_IPADDR      10.0.0.110
-#define CONFIG_SERVERIP      10.0.0.1
-/* #define CONFIG_BOOTFILE      "elinos-lart" */
-/* #define CONFIG_BOOTCOMMAND   "tftp; bootm" */
+#define CONFIG_BOOTARGS      "console=ttySAC0 init=/linuxrc"
+#define CONFIG_ETHADDR      08:00:3e:26:0a:5b
+#define CONFIG_NETMASK      255.255.255.0
+#define CONFIG_IPADDR      192.168.1.128
+#define CONFIG_SERVERIP      192.168.1.120
```




```

#define CONFIG_BOOTFILE      "uImage"
#define CONFIG_BOOTCOMMAND   "nand read 30080000 30000 300000; bootm 30080000"

#if (CONFIG_COMMANDS & CFG_CMD_KGDB)
#define CONFIG_KGDB_BAUDRATE 115200 /* speed to run kgdb serial port */
@@ -108,7 +111,7 @@
    * Miscellaneous configurable options
    */
#define CFG_LONGHELP /* undef to save memory */
-#define CFG_PROMPT "SMDK2410 # " /* Monitor Command Prompt */
+#define CFG_PROMPT "HY2410>" /* Monitor Command Prompt */
#define CFG_CBSIZE 256 /* Console I/O Buffer Size */
#define CFG_PBSIZE (CFG_CBSIZE+sizeof(CFG_PROMPT)+16) /* Print Buffer Size */
#define CFG_MAXARGS 16 /* max number of command args */
@@ -150,6 +153,7 @@

#define CFG_FLASH_BASE      PHYS_FLASH_1

+#if 0
/*-----
    * FLASH and environment organization
    */
@@ -177,5 +181,55 @@

#define CFG_ENV_IS_IN_FLASH 1
#define CFG_ENV_SIZE 0x10000 /* Total Size of Environment Sector */
+#endif
+
+
+#define CFG_NO_FLASH 1
+
+/* Nand Flash Setting */
+#define ONFCONF 0x00
+#define ONFCMD 0x04
+#define ONFADDR 0x08
+#define ONFDATA 0x0C
+#define ONFSTAT 0x10
+#define ONFECC 0x14
+
+#define CONFIG_SKIP_RELOCATE_UBOOT
+#define CONFIG_S3C2410_NAND_BOOT
+
+#define CFG_ENV_IS_IN_NAND 1
+#define CMD_SAVEENV 1
+#define CFG_ENV_OFFSET 0x20000
+#define CFG_ENV_SIZE 0x10000
+
+#define CFG_UBOOT_SIZE 0x30000
+
+#if (CONFIG_COMMANDS & CFG_CMD_NAND)
+#define CFG_NAND_LEGACY 1

```



```

#define CFG_MAX_NAND_DEVICE 1 /* Max number of NAND devices */
#define SECTORSIZE 512
#define ADDR_COLUMN 1
#define ADDR_PAGE 2
#define ADDR_COLUMN_PAGE 3
#define NAND_ChipID_UNKNOWN 0x00
#define NAND_MAX_FLOORS 1
#define NAND_MAX_CHIPS 1
+
#define NAND_WAIT_READY(nand) NF_WaitRB()
+
#define NAND_DISABLE_CE(nand) NF_SetCE(NFCE_HIGH)
#define NAND_ENABLE_CE(nand) NF_SetCE(NFCE_LOW)
+
#define WRITE_NAND_COMMAND(d, adr) NF_Cmd(d)
#define WRITE_NAND_COMMANDW(d, adr) NF_CmdW(d)
#define WRITE_NAND_ADDRESS(d, adr) NF_Addr(d)
#define WRITE_NAND(d, adr) NF_Write(d)
#define READ_NAND(adr) NF_Read()
/* the following functions are NOP's because S3C24X0 handles this in hardware */
#define NAND_CTL_CLRALE(nandptr)
#define NAND_CTL_SETALE(nandptr)
#define NAND_CTL_CLRCLE(nandptr)
#define NAND_CTL_SETCLE(nandptr)
+
#endif /* CONFIG_COMMANDS & CFG_CMD_NAND */

#endif /* __CONFIG_H */

```

这里主要修改了各种配置信息，另外重要的一点是去掉了 **NOR Flash** 的相关配置，增加了 **NAND Flash** 的相关配置。一般来说，其中的 **CONFIG_XXX** 宏用于配置 U-boot 的操作特性，而 **CFG_XXX** 宏则用于配置硬件相关的特性。

八、lib arm/armlinux.c

```
@@ -86,9 +86,7 @@
    image_header_t *hdr = &header;
    bd_t *bd = gd->bd;

-#ifdef CONFIG_CMDLINE_TAG
    char *commandline = getenv ("bootargs");
-#endif

    theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep);

@@ -253,6 +251,14 @@
    setup_videolfb_tag ((gd_t *) gd);
#endif
    setup_end_tag (bd);
+} else
+{
```



```

+     struct param_struct *params = (struct param_struct *)bd->bi_boot_params;
+     memset(params, 0, sizeof(struct param_struct));
+     params->ul.s.page_size = 0x00001000;
+     params->ul.s.nr_pages = (0x04000000 >> 12);
+     strcpy(params->commandline, commandline);
+ }
+ #endif

/* we assume that the kernel is in place */

```

这里主要增加了设置内核引导参数的代码。

7.4.2 配置和编译

按前述内容修改源码之后，HY2410A 的主要硬件接口已经可以使用，如内存、NAND Flash、以太网和串口等，并且可以引导 Linux 内核。下面介绍配置和编译的过程。

由于我们直接在 SMDK2410 目标板的代码上进行了修改，所以可用下面的命令进行配置：

```
make smdk2410_config
```

这样实际上采用的是 SMDK2410 目标板的配置。然后用如下命令开始编译：

```
make
```

编译成功后在源码目录下将出现 u-boot.bin 文件，它就是 U-boot 的二进制映像文件。

下面我们来分析一下配置时的操作，在源码目录下的 Makefile 中有如下几行：

```

SRCTREE      := $(CURDIR)
MKCONFIG     := $(SRCTREE)/mkconfig
# 省略若干行
smdk2410_config :      unconfig
                     @$(MKCONFIG) $(@:_config=) arm arm920t smdk2410 NULL s3c24x0

```

其中变量 CURDIR 代表的是当前目录，而 \$(@:_config=) 是将 \$@ 变量中的 _config 替换为空，可见实际进行的操作为：

```
./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0
```

这里 mkconfig 是源码目录下的一个脚本，其各个参数的含义解释如下。

- ◆ 第一个参数为目标板的名称。
- ◆ 第二个参数为 CPU 架构，这里是 arm。
- ◆ 第三个参数为 CPU 架构下的版本，这里是 arm920t。
- ◆ 第四个参数为目标板，这里是 smdk2410。
- ◆ 第五个参数为厂家名称，它不影响编译过程，故忽略。
- ◆ 第六个参数为 SoC 名称，这里是 s3c24x0。

除第一个参数外，其他 5 个参数将被作为变量写入 include/config.mk 文件，变量名分别为



ARCH, CPU, BOARD, VENDOR, SOC。这个文件在 Makefile 的开头被包含进来，所以影响到了 Makefile 中变量的取值。

编译时所用的交叉编译器可由环境变量 CROSS_COMPILE 指定，如果未定义，则由 Makefile 中的如下语句决定交叉编译器：

```
ifndef CROSS_COMPILE
# 省略若干行
ifeq ($(ARCH),arm)
CROSS_COMPILE = arm-linux-
# 省略若干行
Endif
```

U-boot 的链接地址则写在 board/smdk2410/config.mk 文件中，内容如下：

```
TEXT_BASE = 0x33F80000
```

链接的顺序由链接脚本 board/smdk2410/u-boot.lds 决定，从中可以清楚地看到，首先链接的一个文件是 cpu/arm920t/start.o。

U-boot 编译好之后的映像可以写入目标板运行。如果目标板已经烧写了 U-boot，则可以在 U-boot 环境中下载新的 U-boot 映像并写入 NAND Flash，如：

```
tftp 0x30000000 u-boot.bin
nand erase 0x0 0x30000
nand write 0x30000000 0x0 0x30000
```

如果目标板是裸板，则必须通过 JTAG 接口烧写 U-boot 映像。这要用到三星原厂提供的 sjf2410 工具，注意它是一个 Windows 程序。烧写的命令如下：

```
sjf2410 /f:u-boot.bin
```



Part

第 3 部分 Linux 系统编程

第 8 章 Linux 系统编程基础

第 9 章 Linux 文件系统编程

第 10 章 深入理解进程

第 11 章 socket 编程

第 12 章 多线程并发程序设计

3

第 8 章 Linux 系统编程基础

本书所说的系统编程，是指直接利用系统的底层接口进行编程，以区别于在其他编程平台上进行的应用开发。这些底层的编程接口一般是每个系统必备的，并且往往是直接基于系统调用而实现的，它们是内核对应用程序的直接支持。进行系统编程需要详细理解各种 API（Application Programming Interface，应用编程接口）的用法，以及藏在 API 背后的内核工作机制，这是与其他应用编程不同的要求。

在嵌入式系统中，由于资源的限制，不可能像 PC 机那样装一个大而全的操作系统，以及各种高级开发平台，因此直接面向系统 API 的编程在开发中占有很大的比重。

这一章将主要介绍进行系统编程所需的一些基本概念，为以后介绍各种系统编程接口的章节打下基础。

8.1 系统调用与 API

如图 8.1 所示是 Linux 系统上软件的层次结构。所有的软件都可划分为用户态和内核态两个部分。

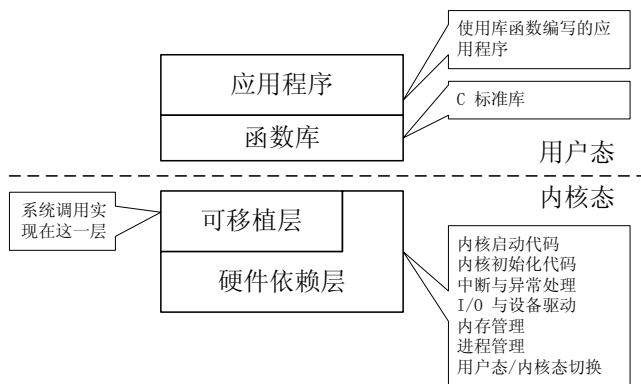


图 8.1 Linux 系统上的软件层次关系

内核是操作系统的核心，内核的代码都运行在内核态。内核的最上层是 Linux 操作系统的与硬件无关的可移植层。通过这一层次，内核可以提供与硬件无关的标准的功能接口给用户态程序，这些接口就是所谓的系统调用。内核的最下层则是与具体硬件有关的硬件依赖层，对于不同的硬件平台，这一层的代码实现也是不同的。

一类重要的系统调用接口是内核提供的通用 I/O 接口。在用户态程序中，是没有权限直接对硬件设备进行操作的，任何对设备的 I/O 操作都必须通过相关的系统 I/O 调用接口进行。此外各种系统应用服务，如多进程、网络等，都是通过系统调用接口提供给用户态程序的。可以这样说，

如果没有系统调用，那么应用程序几乎不能实现有用的功能。

Linux 内核所提供的服务可大致分类如下：

- ◆ 进程控制
- ◆ 文件系统控制
- ◆ 系统控制
- ◆ 内存管理
- ◆ 网络管理
- ◆ 用户管理
- ◆ 进程间通信

如果直接使用系统调用，则还需要少量与硬件平台相关的代码以实现从用户态到内核态的切换，因此应用程序一般通过一个中间层，即 C 标准库来使用内核的系统服务。C 标准库可以直接与应用程序的代码进行链接，这样编程接口就完全与硬件平台无关了，应用程序对于不同平台的可移植性就大大增强了。

用户态与内核态的区分是由 CPU 的工作模式决定的。在用户态下，指令的执行是受到限制的，很多需要特权的指令不允许执行，如果这些指令出现，则会引发异常而陷入内核态。在内核态下，CPU 支持的所有指令都是可以执行的。由于有这种硬件上的支持，内核可以进行任何操作，而应用程序必须通过系统调用才能间接地访问硬件设备。需要指出的是，用户态与内核态只是一种理论上的划分，具体的某种 CPU 可能拥有多个工作模式，如 ARM 的用户模式对应着用户态，而其他模式都属于内核态。

系统调用接口一般通过 CPU 的软中断指令实现从用户态到内核态的调用，这是一种由应用程序主动发起的模式切换。软中断处理完毕返回后重新切换回用户态，实现系统调用的返回。各个系统调用由系统调用号标识，在不同的平台上，向内核传递系统调用号的方式也不同，有的放在寄存器中，有的直接放在指令上。

所谓 API 指的是编写应用程序时所使用的接口，即 C 标准库提供的编程接口。有时也泛指其他编程平台提供的甚至非 C 语言的接口。API 和系统调用并没有一一对应的关系，C 标准库提供的有些功能根本不需要请求内核的服务就可以完成，如内存的复制、数学上的计算等，因此也没有必要发起系统调用。而有些 API 可能需要连续多次进行系统调用。

系统调用在执行完成后会得到一个整型的返回值，按照 Linux 内核的惯例，一般这个值为负数代表有错误发生，负数的绝对值就是错误码。但经过 C 标准库的封装后，很多 API 使用返回值 -1 代表有错误发生，而把错误码放在 `errno` 全局变量中。

下面是一个应用程序使用 API 的例子：

```
/* 文件名: api_test.c */
/* 说明: API 使用例程 */

#include <stdio.h>
#include <unistd.h>
#include <math.h>

int main()
{
    double x = 8.0;
    double y;
    y = sqrt(x);
```

```
printf("square root of %f is %f.\n", x, y);
write(STDOUT_FILENO, "hello!\n", sizeof("hello!\n"));
return 0;
}
```

这个程序因为使用了数学函数，所以编译时要与数学库链接，即用如下命令：

```
gcc api_test.c -o test -lm
```

例程中用到了三个 API: `sqrt`, `printf` 和 `write`。它们与系统调用的关系可由图 8.2 解释。`sqrt` 函数只进行数学计算，不需要内核的参与，因此它虽然是对 C 标准库 API 的调用，但不涉及到系统调用。`printf` 函数的实现实际上要用到 `write` 函数，而 `write` 函数要向标准输出写入数据必须使用内核的文件系统调用。当然，`write` 函数也是一个直接面向应用程序的 API，可由应用程序直接调用。

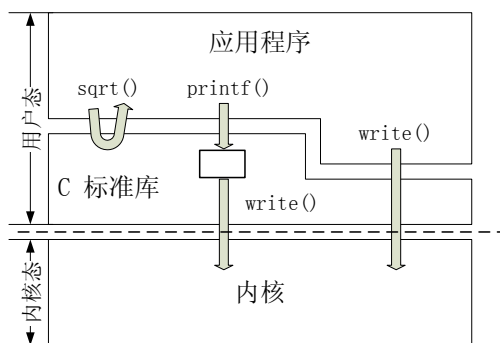


图 8.2 API 与系统调用的关系

8.2 程序的生成与执行

一般情况下不可能直接用机器指令来编写程序，所以要用各种编程语言（如汇编语言、C 语言等）进行编程。从 C 语言转化为机器可识别的指令需要经过预处理、编译、汇编几个过程，生成目标文件；汇编语言则只需要预处理和汇编过程就可以生成目标文件。生成的目标文件并不是直接可以执行的文件，还需要进行链接操作，包括目标文件之间的相互链接以及与系统函数库的链接。这里有一个重要概念就是程序的启动代码，它是所有程序运行时的入口，而 C 程序的 `main` 函数实际上是被启动代码调用的一个函数。

链接生成的可执行程序不仅包含机器指令，还包含一些额外的信息以辅助系统加载和运行程序。直观上，在 Shell 中输入文件名就可以运行一个程序，这个过程实际上包含以下几个重要步骤。

- step 1** Shell 得到输入，发现用户的意图是执行程序。
- step 2** Shell 发起系统调用，产生一个新的进程。
- step 3** 新进程发起系统调用，加载可执行程序并运行。
- step 4** 如果是前台执行，则 Shell 等待新进程的结束，否则 Shell 继续执行。

如图 8.3 所示是一个程序从源码到执行的过程示意。

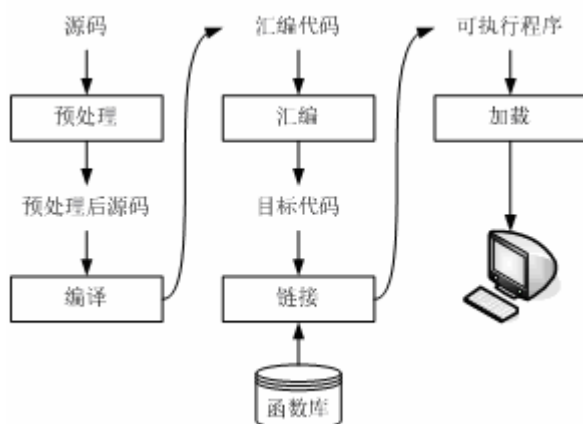


图 8.3 程序的生成与执行

8.3 API 的错误处理

对 API 调用的错误进行处理是系统编程中重要的一环。API 调用中发生的错误一般会在函数返回后通过变量 `errno` 体现出来。`errno` 是一个整型值，用于表示错误的类型，它的接口头文件及定义如下：

```
#include <errno.h>
extern int errno;
```

在代码中判断错误类型时错误码很有用，但如果用于输出则不太直观。C 标准库提供了一个函数用于将错误码转换为一个描述错误的字符串，其接口头文件及函数原型如下：

```
#include <string.h>
char *strerror(int errnum);
```

其中参数 `errnum` 是错误码，返回值是描述错误类型的字符串。

更方便的做法是直接调用 `perror` 函数将当前的 `errno` 变量对应的错误信息输出到标准错误输出（控制台屏幕）上，它的接口头文件及函数原型如下：

```
#include <stdio.h>
void perror(const char *s);
```

它会首先将字符串 `s` 输出，然后再输出对错误的描述。

8.4 命令行参数与环境变量

`main` 函数是 C 语言应用程序的入口函数。系统在执行程序时，会将命令行参数及当前 Shell 的环境变量通过函数参数传入。`main` 函数的一个完整原型如下：

```
int main(int argc, char *argv[], char *envp[]);
```

其中参数 `argc` 和 `argv` 用来传入命令行参数，`envp` 用来传入环境变量。`main` 函数的返回值将是整个应用程序的返回值。如果不需要环境变量，则 `envp` 参数可以没有，如果连命令行参数都不需要，则 `main` 函数可以没有参数。

`argc` 参数表示的是命令行参数的个数，而 `argv` 参数则指向一个字符串数组，其中每个字符串是一个命令行参数，同时，`argv` 数组的最后一个元素是 `NULL`，表示数组的结束。需要注意的是，Shell 认为命令中用空格隔开的各个部分都是参数，包括可执行文件名本身。例如下面这条命令：

```
./test -n 3.4
```

它传入 `main` 函数的 `argc` 参数的值为 3，而 `argv` 参数在内存中的布局如图 8.4 所示。

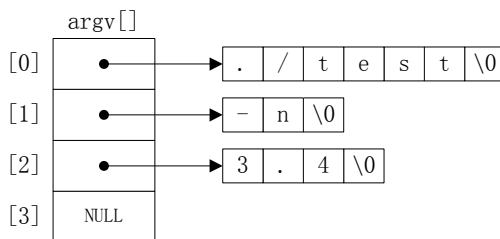


图 8.4 命令行参数传入方式

与 `argv` 类似，`envp` 也是一个以 `NULL` 指针结束的字符串数组，其中的每个字符串表示一个环境变量，表达成“变量名=值”的形式。

使用命令行参数与环境变量的一个例子如下：

```
/* 文件名: arg_env.c */
/* 说明: 命令行参数与环境变量例程 */

#include <stdio.h>

int main(int argc, char *argv[], char *envp[])
{
    int i;
    /* 输出所有命令行参数 */
    for (i = 0; i < argc; i++) {
        printf("Para %d: %s\n", i, argv[i]);
    }
    /* 输出所有环境变量 */
    for (i = 0; envp[i]; i++) {
        printf("Env %d: %s\n", i, envp[i]);
    }
    return 0;
}
```

将上述源码编译成可执行文件，假定文件名为 `test`，以如下命令执行：

```
./test -n 3.4
```

则输出结果为：

```
Para 0: ./test
Para 1: -n
Para 2: 3.4
Env 0: TERM=linux
Env 1: SHELL=/bin/bash
```

输出的环境变量随运行环境的不同而异。



第 9 章 Linux 文件系统编程

对用户使用操作系统来说，文件系统有着决定性的意义。理论上，内核没有文件系统也能运行起来，甚至可以执行应用程序，但是，这样的操作系统基本是无用的，因为缺乏与用户的交互。操作系统的可用性取决于对文件系统的支持，操作系统是通过文件系统接口提供各种功能的，并且文件系统也是组织系统资源的一种方式。

本章将主要从应用的角度来讨论文件系统，介绍 Linux 文件系统的基本概念及常用的编程接口。

9.1 文件的概念

文件在 Linux 系统中是一个很通用的概念，它是对系统资源的一个抽象，是对系统资源进行访问的一个通用接口，诸如内存、硬盘、一般设备及进程间通信的通道这些系统资源都可表述为文件。这样做可以对这些资源提供通用的操作接口，极大地简化了系统编程接口的设计。因此经常说，在 Linux 系统上，一切皆文件，文件无所不在。以下将具体介绍几种可以通过文件来访问的系统资源，也是文件的几种类型。

一、普通文件

普通文件就是一般意义上的文件，它们作为数据存储的系统磁盘中，可以随机访问文件的内容。Linux 系统中的文件是面向字节的，文件的内容以字节为单位进行存储与访问。文件系统本身并不关心文件的内容，只是提供数据存储与访问的透明通道。文件内容的含义则是由应用程序解释的。

二、目录

在 Linux 系统中，目录也是一种特殊的文件，它们用来包含文件，文件一定在某个目录下。目录可以像普通文件一样打开、关闭，以及进行相应的操作，但为了操作方便，系统对于目录也提供了一些特定的操作接口。

三、管道

管道是 Linux 系统中一种进程间通信的机制。通常，一个进程写一些数据到管道中，这些数据就可以被另一个进程从这个管道中读取出来。编程接口与上面所说的普通文件是相同的。管道可以分为两种类型：无名管道与命名管道。

- ◆ 无名管道由进程在使用时创建，读写结束关闭文件后就消失。之所以称为无名管道，是因为它们并不存在于文件系统中，没有文件名称。
- ◆ 命名管道在形式上就是文件系统中的文件，虽然并不占用存储文件内容的磁盘空间，但有自己的文件名。因此使用这个名称可以在两个独立的进程间进行通信，而无名管道却只能在有亲缘关系的进程间进行通信。命名管道也常称为 FIFO (First In First Out, 先进先出)，因为数据在管道中传送的方式是先进先出，即先写入管道的数据被先读取出来。

四、设备文件

设备文件形式上也是文件系统上的文件，与普通文件不同的是，它没有具体的内容，对设备文件的读写操作实际上与某个设备的输入输出操作关联在一起。按照惯例，设备文件都建立在 `/dev` 目录下，但并没有实质上的限制。设备文件有两种类型：字符设备文件和块设备文件，分别对应着字符设备和块设备。

- ◆ 字符设备能够以字符（一个字节）为单位进行输入输出操作，内核不会对设备输入输出的数据进行缓冲和排序。
- ◆ 块设备的输入输出以块为单位，每个块有固定的字节数（一般为 512 字节的整数倍）并且有唯一的地址，可以进行随机访问。块设备的最大特点就是可以容纳一个文件系统，有文件系统的块设备可以被挂载到某个目录中。对块设备的访问将被内核缓冲并且有可能重新编排访问请求的顺序，以提高数据的读写效率。

五、符号链接

符号链接是一种特殊的文件，它的内容是指向另一个文件的路径。当对符号链接进行操作时，系统根据情况会将这个操作转移到它所指向的文件上去，而不是对它本身进行操作。例如，读一个符号链接时，实际读到的是它所指向的文件的内容。

六、socket

socket（或称套接字）也是一种进程间通信的方式。与管道不同的是，它们可以在不同主机上的进程间通信，实际上就是网络通信。socket 在 Linux 系统上也是以文件的方式进行操作的。

本章中将主要以普通文件为例对文件的编程接口进行介绍，但读者应意识到，它们对于其他类型的文件也是适用的。

9.2 文件描述符与索引节点

一般来讲，使用与管理文件是通过文件名来进行的，但从应用编程的角度来看，文件描述符更有用，而系统中的文件在本质上是通过其索引节点进行管理的。

文件描述符是应用程序中表示被打开的文件的一个整数，其他对文件的操作接口都要使用这个整数来指定所操作的文件。

从系统的角度来看，文件的索引节点（inode）是文件的唯一标识。一个文件的 inode 包含文件系统处理文件所需要的全部信息，如访问权限、当前大小等。如果详细说明，则实际上存在两种类型的 inode：一个是所谓的内核 inode（in-core indoe），保存在内存中，在系统中每个打开的文件都对应着一个内核 inode；另一个是磁盘 inode（on-disk inode），在文件系统上的每一个文件都有一个磁盘 inode，保存在磁盘上，它所保存的具体信息与文件系统的类型有关。当进程打开一个文件时，文件的磁盘 inode 中的信息就会被载入内存，并建立一个内核 inode。当内核 inode 被修改后，系统负责将其同步到磁盘上。磁盘 inode 与对应的内核 inode 所保存的信息并不是完全相同的。内核 inode 记录的是关于文件的更通用的一些信息，而忽略掉与具体文件系统类型相关的一些信息。

一般而言，一个 inode 应当记录如下信息。

- ◆ 文件类型。
- ◆ 与文件相关的硬链接的个数。
- ◆ 以字节为单位的文件的长度。
- ◆ 设备标识符（即包含文件的设备的标识符）。
- ◆ 在文件系统中标识文件的索引号。
- ◆ 文件所属用户的 UID（User ID，用户标识符）。
- ◆ 文件所属组的 GID（Group ID，组标识符）。
- ◆ 各种时间戳，包括文件状态的改变时间、文件的最后访问时间和最后修改时间。

9.3 文件操作的系统调用接口

前面我们已经了解到，文件是 Linux 系统中的重要概念。它不仅是对普通文件的操作接口，也是设备通信、进程间通信、网络通信的重要编程接口。因此文件操作的相关系统调用也是 Linux 内核提供的最重要的编程接口。

本节将重点叙述如下几个常用的文件操作系统调用。

- ◆ open：打开文件。
- ◆ read：从已打开的文件中读取数据。
- ◆ write：向已打开的文件中写入数据。
- ◆ close：关闭已打开的文件。
- ◆ ioctl：向文件传递控制信息或发出控制命令。

对文件进行操作的一般过程是这样的：先打开文件，内核对打开的文件进行管理，打开成功后应用程序将获得文件描述符；然后应用程序使用文件描述符对文件进行读写操作；当全部操作完毕后，应用程序需要将文件关闭以释放用于管理打开文件的内存。

文件描述符是一个取值从 0 开始的整数。内核默认一个进程同时打开的文件数有一个上限，也就是文件描述符取值的上限，一般是 1024。

每个进程在启动后就默认有三个打开的文件描述符 0、1 和 2，如果启动程序时没有进行重定向，则文件描述符 0 关联到标准输入，1 关联到标准输出，2 关联到标准错误输出。在 C 库函数中可以使用以下几个宏来表示这几个文件描述符：

```
#define STDIN_FILENO 0    /* 标准输入 */
#define STDOUT_FILENO 1   /* 标准输出 */
#define STDERR_FILENO 2   /* 标准错误输出 */
```

本节的主要内容是介绍这些编程接口的使用。

9.3.1 打开文件

在访问文件之前，首先应该打开文件。可以使用 open 或 creat 函数来打开文件，它们的接口头文件及函数原型如下：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

其各个参数及返回值的含义解释如下。

- ◆ **pathname**: 要打开的文件名称。
- ◆ **flags**: 标志位, 指定打开文件的操作方式及打开时的一些行为。
- ◆ **mode**: 用于指定新文件的权限标志位。
- ◆ **返回值**: 操作成功则返回文件描述符, 否则返回 -1 并设置变量 **errno** 的值。

flags 参数有以下几个基本的取值。

- ◆ **O_RDONLY**: 以只读方式打开文件。
- ◆ **O_WRONLY**: 以只写方式打开文件。
- ◆ **O_RDWR**: 以读写方式打开文件。

这几个标志指定打开文件的操作方式, 它们是互斥的, 不能同时使用, 但可以与下述标志用按位或的方式组合起来使用。

- ◆ **O_CREAT**: 如果被打开的文件不存在, 则自动创建这个文件。
- ◆ **O_EXCL**: 如果 **O_CREAT** 标志已经使用, 那么当由 **pathname** 参数指定的文件已存在时 **open** 函数返回失败。如果 **pathname** 给出的是一个符号链接, 无论它指向的文件是否存在, 对 **open** 函数的调用都会返回失败。
- ◆ **O_NOCTTY**: 如果被打开的文件是一个终端设备文件 (如 **/dev/tty**), 它不会成为这个进程的控制终端。
- ◆ **O_TRUNC**: 如果被打开的文件存在并且是以可写的方式打开的, 则清空文件原有的内容。
- ◆ **O_APPEND**: 新写入的内容将被附加在文件原来的内容之后, 即打开后文件的读写位置被置于文件尾。
- ◆ **O_NONBLOCK**: 被打开的文件将以非阻塞的方式进行操作。
- ◆ **O_NDELAY**: 同 **O_NONBLOCK**。
- ◆ **O_SYNC**: 被打开的文件将以同步 I/O 的方式进行操作, 即任何写操作都会先被同步到硬件设备上。同步完成后, 对写函数的调用才返回。
- ◆ **O_NOFOLLOW**: 如果 **pathname** 是一个符号链接, 则对 **open** 函数的调用将返回失败。
- ◆ **O_DIRECTORY**: 如果 **pathname** 不是目录, 则对 **open** 函数的调用将返回失败。

需要注意的是, **open** 函数有两个原型, 其中一个多出了参数 **mode**, 它用于指定创建的新文件的访问权限。如果打开时使用了 **O_CREAT** 标志创建新文件, 则一般都要给出 **mode** 参数, 它的一些常用取值如表 9.1 所示, 这些值可以用按位或的方式组合使用。新文件的所属用户和所属组则是创建它的进程的所属用户和所属组。



表 9.1 常用权限标志位

权限标志定义	对应的八进制形式	含义
S_IRWXU	00700	文件所属用户有读写和执行权限
S_IRUSR (S_IREAD)	00400	文件所属用户有读权限
S_IWUSR (S_IWRITE)	00200	文件所属用户有写权限
S_IXUSR (S_IEXEC)	00100	文件所属用户有执行权限
S_IRWXG	00070	组内用户有读写和执行权限
S_IRGRP	00040	组内用户有读权限
S_IWGRP	00020	组内用户有写权限
S_IXGRP	00010	组内用户有执行权限
S_IRWXO	00007	其他用户有读写和执行权限
S_IROTH	00004	其他用户有读权限
S_IWOTH	00002	其他用户有写权限
S_IXOTH	00001	其他用户有执行权限

鉴于在调用 `open` 函数时, `O_WRONLY`, `O_CREAT`, `O_TRUNC` 三个标志位经常组合使用, 因此由一个专门的函数 `creat` 来实现。对 `creat` 函数的如下调用:

```
creat(pathname, mode);
```

实际上等价于:

```
open(pathname, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

这两个函数在打开文件成功时将返回一个文件描述符, 可用于随后的 `read/write` 或其他对文件的操作使用。两个不同的进程打开同一个文件是允许的, 但它们得到的文件描述符一般是不同的。如果它们都对文件进行写操作, 就会出现数据不一致的情况, 也就是最后写入的可能覆盖先前其他进程写入的内容, 这就涉及到进程间数据共享和同步的概念了。

如果打开操作失败, 这两个函数会返回 `-1`, 并将 `errno` 变量设置为一个合适的错误值。

9.3.2 从文件读取数据

文件打开以后就可以进行读写操作了。读操作的接口头文件及函数原型如下:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

其各个参数及返回值的含义解释如下。

- ◆ `fd`: 要读取的文件的描述符。
- ◆ `buf`: 指向读取到的数据要放入的缓冲区。
- ◆ `count`: 要读取的字节数 (缓冲区大小)。
- ◆ 返回值: 读取到的字节数, 失败则返回 `-1` 并设置变量 `errno` 的值。



这里的 `size_t` 型实际上就是无符号整型，而 `ssize_t` 型就是有符号整型。

这个函数将从 `fd` 代表的文件的当前读写位置读取不超过 `count` 个字节到 `buf` 指向的内存中，并返回读到的字节数。



`buf` 指向的内存空间必须事先分配好。

对于普通文件来说，读操作完成后，文件的读写位置会向后移动，移动的长度就是读取的字节数，下一次读操作将从新的读写位置开始。

`read` 函数的返回值小于指定的 `count` 是可能的，并不是错误。出现这种情况有各种原因，比如，文件本身可供读取的字节数比 `count` 小或者 `read` 系统调用被信号打断等。`read` 系统调用看似简单，但实际上对于各种可能情况的处理是比较复杂的，因为 I/O 操作有很多异常情况要考虑到。下面列出了 `read` 系统调用中可能遇到的情况及其处理方法。

- ◆ 调用返回值等于 `count`，读取的数据存放在 `buf` 指向的内存中，结果与预期一致。
- ◆ 调用返回一个大于 0 小于 `count` 的值，读取的字节数存放在 `buf` 指向的内存中。出现这种情况可能是一个信号打断了读取过程，或在读取中发生了一个错误，或读取的有效字节数大于 0 但不足 `count` 个，或在读入 `count` 个字节前文件已经结束。如果读取的过程被信号打断则可以再次进行读取。
- ◆ 调用返回 0，说明文件已结束，没有可以读入的数据。
- ◆ 调用阻塞，说明没有可读的数据，这种情况下如果以非阻塞方式操作文件，那么会立即返回错误。
- ◆ 调用返回 -1，并且 `errno` 变量被设置为 `EINTR`，表示在读入有效字节前收到一个信号，这种情况可以重新进行读操作。
- ◆ 调用返回 -1，并且 `errno` 变量被设置为 `EAGAIN`，这说明是在非阻塞方式下读文件，并且没有可读的数据。
- ◆ 调用返回 -1，并且 `errno` 变量被设置为非 `EINTR` 或 `EAGAIN` 的值，表示有其他类型的错误发生，必须根据具体情况进行处理。

由于有各种异常情况的存在，为了从文件中可靠地读取指定的字节数，就必须对这些情况进行处理，必要时需重新进行读操作。这对于设备文件、管道或 `socket` 来说尤其有意义。例如，用下面的代码能够可靠地从文件中读取指定的字节数（假设文件是以阻塞的方式进行操作的）：

```
ssize_t ret;
while (len != 0 && (ret = read(fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR) continue;
        perror("read");
        break;
    }
    len -= ret;
    buf += ret;
}
```

在上述代码中，变量 `len` 的初始值是要读取的字节数，`buf` 的初始值则指向用来存放数据的缓冲区，`fd` 是要读取的文件的描述符。这里把读操作放在循环中，当一次读操作没有得到 `len` 个字节时，将调整 `len` 和 `buf` 的值继续进行读操作。如果读操作返回 `-1`，说明有错误发生，这时如果错误码是 `EINTR`，说明只是被信号打断，可以继续读，其他情况则被认为是严重的错误，不能再继续读。

以上是以阻塞方式读文件的例子，这时，如果文件是一个设备文件并且设备没有可读的数据，进程将进入睡眠状态不再继续执行，或者说阻塞在 `read` 系统调用处，直到设备有了可读的数据才会被唤醒继续执行。很多时候我们需要进程能够立刻返回以处理其他事务，那么就需要采用非阻塞的方式来操作文件，举例如下：

```
ssize_t nr;
start:
nr = read (fd, buf, len);
if (nr == -1) {
    if (errno == EINTR) goto start;
    if (errno == EAGAIN)
        /* 处理其他事务，在恰当时再调用 read */
    else
        /* 有错误发生，处理错误 */
}
```

可以看到，在采用非阻塞的方式读文件时，如果读操作返回为 `-1`，我们必须检查错误码是否为 `EINTR` 或 `EAGAIN`，如果是 `EINTR`，可以再次进行读操作，而如果是 `EAGAIN`，表示要读取的（设备）文件现在没有可供读取的数据，因此进程可以继续处理其他事务，而后在恰当的时机再来读取这个文件。

9.3.3 写数据到文件

向打开文件中写入数据的接口头文件及函数原型如下：

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

其各个参数及返回值的含义解释如下。

- ◆ `fd`：要写入的文件的描述符。
- ◆ `buf`：指向要写入的数据所存放的缓冲区。
- ◆ `count`：要写入的字节数。
- ◆ 返回值：实际写入的字节数，失败则返回 `-1`，并设置变量 `errno` 的值。

这个函数会从 `fd` 所代表的文件的当前读写位置开始，把 `buf` 指向的内存中最多 `count` 个字节写入文件。写入成功则返回写入的字节数，并更新文件的读写位置。

`write` 系统调用返回大于 `0` 而小于 `count` 的值是合法的，并不表示有错误发生。与 `read` 系统调用一样，`write` 系统调用中也可能发生各种各样的异常，应用程序可根据返回值和 `errno` 变量所表示的错误码进行处理，一些可能的情况如下。

- ◆ 调用返回值等于 `count`，说明数据全部写入成功。
- ◆ 调用返回一个大于 0 小于 `count` 的值，说明部分数据没有写入。这可能是因为写入过程被信号打断，或者底层的设备暂时没有足够的空间存放所写入的数据。
- ◆ 调用阻塞，说明暂时不能写入数据，这种情况下如果以非阻塞方式操作文件，那么会立即返回错误。
- ◆ 调用返回 -1，并且 `errno` 变量被设置为 `EINTR`，表示在写入一个有效字节前收到一个信号，应用程序可以再次进行写操作。
- ◆ 调用返回 -1，并且 `errno` 变量被设置为 `EAGAIN`，说明是在非阻塞方式下写文件但文件暂时不能写入数据。
- ◆ 调用返回 -1，并且 `errno` 变量被设置为 `EBADF`，表示给定的文件描述符非法，或者文件不是以写方式打开的。
- ◆ 调用返回 -1，并且 `errno` 变量被设置为 `EFAULT`，表示 `buf` 是无效指针。
- ◆ 调用返回 -1，并且 `errno` 变量被设置为 `EFBIG`，表示写入的数据超过了最大的文件尺寸，或者超过了允许的文件的读写位置。
- ◆ 调用返回 -1，并且 `errno` 变量被设置为 `EPIPE`，说明写入时发生了数据通道断裂的错误，这种情况只在文件是管道或 `socket` 的情况下发生。这种情况下，进程还将收到一个 `SIGPIPE` 信号，信号的默认处理程序是使进程退出。
- ◆ 调用返回 -1，并且 `errno` 被设置为 `ENOSPC`，说明底层设备没有足够的空间。

与读操作一样，在调用 `write` 函数向文件写数据时，并不能保证一次写完所提供的全部数据，并且也会有各种各样的异常情况，这时需要采取措施以保证数据的可靠写入，这在对设备文件或套接字文件操作时经常会用到，举例如下：

```
ssize_t ret;
while (len != 0 && (ret = write(fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR) continue;
        perror("write");
        break;
    }
    len -= ret;
    buf += ret;
}
```

在上述代码中，变量 `len` 的初始值是要写入的字节数，`buf` 的初始值则指向用来存放数据的缓冲区，`fd` 是要写入的文件的描述符。当一次 `write` 调用没有写完全部数据时，代码中调整了 `len`，`buf` 的值继续进行写操作。这里假定写操作是以阻塞的方式进行的。如果以非阻塞方式进行写操作，则当函数返回 -1 时，必须检测 `errno` 变量的值是否为 `EAGAIN`，以决定能否再进行写操作。

9.3.4 发送控制命令

在 Linux 系统上，那些不能被抽象为读和写的文件操作统一由 `ioctl` 操作代表。`ioctl` 操作用

于向文件发送控制命令，这些命令不能被视为是输入输出数据流的一部分，而只是影响文件的操作方式。对于设备文件来说，`ioctl` 操作常用于修改设备的参数。

`ioctl` 系统调用的接口头文件及函数原型如下：

```
#include <sys/ioctl.h>
int ioctl(int fd, int request, ...);
```

其各个参数及返回值的含义如下。

- ◆ `fd`：要操作的文件的描述符。
- ◆ `request`：代表要进行的操作，不同的（设备）文件有不同的定义。
- ◆ 可变参数：取决于 `request` 参数，通常是一个指向变量或结构体的指针。
- ◆ 返回值：成功返回 0，有些 `ioctl` 操作返回其他非负值，错误返回 -1。

`ioctl` 能够进行的操作根据 `fd` 所代表的文件的具体类型而变化，非常繁多。下面举一个例子，使用 `TIOCGWINSZ` 命令获得终端的窗口大小，其源码如下：

```
/* 文件名：console_size.c */
/* 说明：使用 ioctl 获得控制台窗口的大小 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>

int main(void)
{
    struct winsize size;
    /* 判断标准输出是否为 tty 设备，防止输出被重定向的情况 */
    if (!isatty(STDOUT_FILENO)) return -1;
    /* 获得窗口大小 */
    if (ioctl(STDOUT_FILENO, TIOCGWINSZ, &size) < 0) {
        perror("ioctl TIOCGWINSZ error");
        return -1;
    }
    /* 输出结果 */
    printf("rows is %d, columns is %d\n", size.ws_row, size.ws_col);
    return 0;
}
```

9.3.5 关闭文件

程序完成对文件的操作后，要使用 `close` 系统调用将文件关闭，其接口头文件与函数原型如下：

```
#include <unistd.h>
int close(int fd);
```

其中参数 `fd` 是要关闭的文件的描述符，返回值在操作成功的情况下是 0，否则是 -1。

9.4 标准 I/O 函数库

C 标准库提供了文件的标准 I/O 函数库，相比前述的系统调用，主要差别是实现了一个跨平台的用户态缓冲的解决方案。标准 I/O 函数库使用简单，与系统调用 I/O 相似，也包括打开、读写、关闭这些操作，主要的函数列举如下。

- ◆ 打开与关闭文件：fopen, fclose。
- ◆ 读写文件：fread, fwrite。
- ◆ 读写文本行：fgets, fputs。
- ◆ 格式化读写：fscanf, fprintf。
- ◆ 标准输入输出：printf, scanf。
- ◆ 读写字符：fgetc, getc, getchar, fputc, putc, putchar。
- ◆ 其他：fflush, fseek。

如图 9.1 所示是一个 I/O 缓冲的例子。这里用了 printf 函数向标准输出写入多个字符，所写入的字符被放在一个用户态的缓冲区中，直到碰到一个换行符系统才调用 write 函数将缓冲区中的数据写入标准输出。也就是说，在换行符之前写入的字符并不会立刻出现在控制台屏幕上。

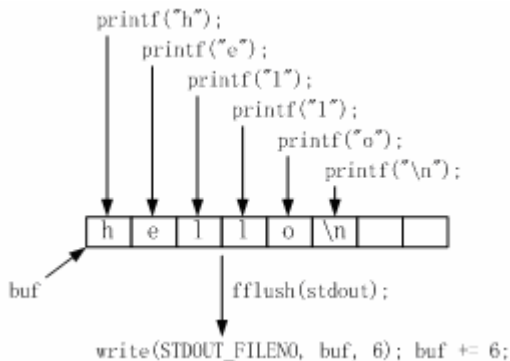


图 9.1 I/O 缓冲示意

这样做的目的是提高系统进行 I/O 操作的效率，因为系统调用要请求内核的服务，会引发 CPU 模式的切换，期间会有大量的堆栈数据保存操作，开销比较大。如果频繁地进行系统调用，会降低应用程序的运行效率。有了缓冲机制以后，多个读写操作可以合并为一次系统调用，减少了系统调用的次数，将大大提高程序的运行效率。

由此可知，所谓标准 I/O 函数实际上是对底层系统调用的包装，最终读写设备或文件的操作仍需调用系统 I/O 函数来完成。它们之间的层次关系如图 9.2 所示。

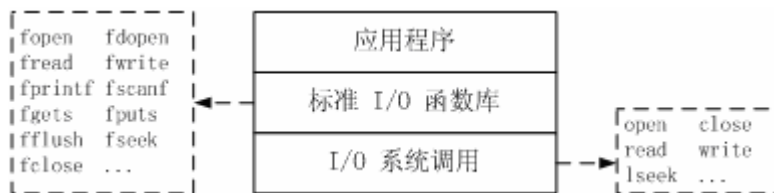


图 9.2 标准 I/O 与系统 I/O 的关系

在具体叙述标准 I/O 库函数的功能前，先了解两个重要概念：文件指针和流。

标准 I/O 函数并不直接操作文件描述符，而是使用文件指针。文件指针与文件描述符是一一对应的关系，这种对应关系由标准 I/O 库自己内部维护。应用程序调用时，只需要提供文件指针即可。文件指针指向的数据类型为 FILE 型，但应用程序无须关心它的具体内容。

在标准 I/O 中，一个打开的文件称为流（stream），流可以用于读（输入流）、写（输出流）或者是读写（输入输出流）。每个进程在启动后就会打开三个流，与打开的三个文件相对应：stdin 代表标准输入流，stdout 代表标准输出流，stderr 代表标准错误输出流，它们都是（FILE *）型的指针。



标准错误输出流不进行缓冲，输出的内容会马上同步到文件（控制台设备）。

与标准 I/O 相关的定义和声明都放在以下头文件中：

```
#include <stdio.h>
```

9.4.1 fopen

fopen 函数用于打开一个文件流，其原型如下：

```
FILE *fopen(const char *filename, const char *mode);
```

其各个参数及返回值的含义解释如下。

- ◆ filename：被打开的文件的名称（可包含路径）。
- ◆ mode：字符串，用于表示打开的模式。
- ◆ 返回值：打开成功后的文件指针，失败则返回 NULL。

用这个函数打开文件，将会创建一个相关联的流（若成功返回，流非空）。其中 mode 参数用于指定打开的方式，它可以是如表 9.2 所示的字符串之一。

表 9.2 打开方式字符串及含义

字符串	含义
r 或 rb	表示以只读方式打开
w 或 wb	表示以只写方式打开，若文件有内容，则清空
a 或 ab	表示以只写方式打开，原内容保留，写入的内容附加在文件流尾部
r+ 或 rb+ 或 r+b	以更新方式打开，此时文件可读可写
w+ 或 wb+ 或 w+b	以更新方式打开，文件可读可写，但打开时清空文件内容
a+ 或 ab+ 或 a+b	以更新方式打开，文件可读可写，写入的内容附加在文件流尾部

9.4.2 fread 和 fwrite

fread 函数用于从打开的文件流中读数据，其原型如下：

```
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

其各个参数及返回值的含义解释如下。

- ◆ ptr: 指向用于存放读取到的数据的缓冲区。
- ◆ size: 被读取的数据块的长度。
- ◆ nitems: 要读取的数据块的个数。
- ◆ stream: 被读取的文件指针。
- ◆ 返回值: 实际读取到的数据块的个数。

使用 `fread` 函数需要注意的是，它以数据块（或称记录）为单位进行读取，返回值也是成功读取的数据块的个数，而不是字节数，这个数目有可能比要读取的个数 `nitems` 少。

`fwrite` 函数用于向打开的文件流写入数据，其原型如下：

```
size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
```

其各个参数及返回值的含义解释如下。

- ◆ ptr: 指向存放写入数据的缓冲区。
- ◆ size: 要写入的数据块的长度。
- ◆ nitems: 要写入的数据块的个数。
- ◆ stream: 要写入的文件指针。
- ◆ 返回值: 实际写入的数据块的个数。

与 `fread` 函数类似，`fwrite` 函数也是以数据块为单位向文件流写入数据的。

9.4.3 fclose

`fclose` 函数用于关闭文件，其原型如下：

```
int fclose(FILE *stream);
```

这个函数可以关闭 `stream` 参数所代表的文件，如果关闭成功则返回 0，否则将返回 EOF 并且设置变量 `errno` 的值以指示错误。关闭前会自动将文件流中的数据写入文件。

9.4.4 fflush

`fflush` 函数用于将文件流中的数据更新到真实的文件中，其原型如下：

```
int fflush(FILE *stream);
```

其中参数 `stream` 是要操作的文件指针，如果更新成功则返回 0，否则将返回 EOF 并且设置变量 `errno` 的值以指示错误。

在用 `fclose` 关闭文件前，系统会自动调用 `fflush` 更新数据。

9.4.5 fseek 和 ftell

`fseek` 用于移动文件流的读写位置，其原型如下：

```
int fseek(FILE *stream, long offset, int whence);
```

其各个参数及返回值的含义如下。

- ◆ **stream**: 被操作的文件指针。
- ◆ **offset**: 读写位置的偏移量。
- ◆ **whence**: 用于指定偏移量的相对起点。
- ◆ **返回值**: 0 表示操作成功, -1 表示操作失败并且 **errno** 变量的值被设置为错误码。

whence 参数的取值及含义如下。

- ◆ **SEEK_SET**: 表示偏移量相对于文件开头。
- ◆ **SEEK_CUR**: 表示偏移量相对于当前的读写位置。
- ◆ **SEEK_END**: 表示偏移量相对于文件末尾。

如果要将读写位置移动到文件流的开头, 还可以用下面这个函数:

```
void rewind(FILE *stream);
```

ftell 函数可以得到文件流的读写位置, 其原型如下:

```
long ftell(FILE *stream);
```

其中参数 **stream** 是文件指针, 返回值就是文件流的当前读写位置 (相对于文件开头)。

9.4.6 fgetc, getc 和 getchar

fgetc 函数用于从文件流中读取一个字符, 其原型如下:

```
int fgetc(FILE *stream);
```

与它功能相同的函数是 **getc**, 原型如下:

```
int getc(FILE *stream);
```

其中 **stream** 参数是要读取的文件流。它们的返回值虽然是整型, 但实际表示的是读到的字符, 只不过进行了类型转换。如果读操作发生错误或者到达文件尾, 则返回值是 **EOF**。

getc 与 **fgetc** 的区别在于它可能是由宏定义实现的, 因此参数可能在宏展开以后被使用多次, 如果参数本身是一个表达式就会被多次求值, 这种情况在使用中应该避免。

getchar 函数用于从标准输入流读取一个字符, 其原型如下:

```
int getchar(void);
```

实际上对 **getchar** 的调用完全等价于以下函数调用:

```
getc(stdin);
```

9.4.7 fputc, putc 和 putchar

fputc 函数用于向文件流写入一个字符, 其原型如下:


```
int fputc(int c, FILE *stream);
```

putc 函数与它的功能相同，原型如下：

```
int putc(int c, FILE *stream);
```

其各个参数及返回值的含义解释如下。

- ◆ c：是要写入的字符，它虽然是整型，但写入时会将其转换为无符号字符型。
- ◆ stream：要写入的文件指针。
- ◆ 返回值：写入的字符转换成整型后的值，发生错误则返回 EOF。

putc 函数与 fputc 函数的区别在于它有可能是用宏定义实现的。

putchar 函数用于向标准输出写入一个字符，其原型如下：

```
int putchar(int c);
```

实际上对 putchar 的调用完全可视为以下函数调用：

```
putc(c, stdout);
```

9.4.8 fgets 和 gets

fgets 用于从文件流中读取一行数据，其原型如下：

```
char *fgets(char *s, int size, FILE *stream);
```

其各个参数和返回值的含义解释如下。

- ◆ s：指向一块缓冲区，用于存放读到的数据。
- ◆ size：读取的字节数上限，实际读取的字节数不会超过 size-1。
- ◆ stream：要读取的文件指针。
- ◆ 返回值：等于 s，如果有错误发生或文件结束，则返回 NULL。

用 fgets 函数读取数据时，当读到一个换行符，或者文件结束，或者读取的字节数达到 size-1，则读操作不再继续，函数返回。fgets 函数还会在读到的数据最后加一个字符 \0，使之变成一个合法的字符串。注意，如果读到换行符，则换行符也在读到的数据中。

gets 函数用于从标准输入读取一行数据，其原型如下：

```
char *gets(char *s);
```

其中参数 s 指向用于存放数据的缓冲区，如果读取成功，则返回值就是 s，否则返回 NULL。



gets 是一个不提倡使用的函数，因为它对读入的字节数没有控制，缓冲区是否会溢出完全取决于用户的输入。

9.4.9 fputs 和 puts

fputs 函数用于向文件流写入一个字符串，其原型如下：

```
int fputs(const char *s, FILE *stream);
```

其各个参数及返回值的含义解释如下。

- ◆ **s**：要写入的字符串，必须是以 `\0` 结尾的合法字符串。
- ◆ **stream**：要写入的文件指针。
- ◆ **返回值**：非负数表示写入成功，有错误发生则返回 `EOF`。

fputs 函数在向文件流写入字符串时，字符串的结束符 `\0` 并不会被写入。

puts 函数将字符串写入标准输出，其原型如下：

```
int puts(const char *s);
```

其中 **s** 参数是要写入的字符串，它的返回值的含义与 **fputs** 函数相同。

与 **fputs** 函数不同的是，**puts** 函数在将字符串写入之后会再写入一个换行符。

9.4.10 fprintf, printf 和 sprintf

fprintf 是向文件流格式化写入数据的函数，其原型如下：

```
int fprintf(FILE *stream, const char *format, ...);
```

其各个参数及返回值的含义解释如下。

- ◆ **stream**：要写入的文件指针。
- ◆ **format**：格式化字符串。
- ◆ **可变参数**：要写入的数据。
- ◆ **返回值**：如果写入成功，则返回格式化后字符串的长度，也就是写入数据的长度，负数表示有错误发生。

使用 **fprintf** 函数最重要的是理解格式化字符串 **format** 的使用。简单地说，**format** 是一个字符串，所包含的内容可以分为两种：一种是普通字符，这些字符将被原样复制到最终输出的字符串中；另一种是以符号 `%` 开头的转换符，函数在解析格式字符串时，每遇到一个转换符，就会从可变参数中取一个或多个参数，以指定格式转换为最终输出的字符串。转换符的书写形式如下：

```
%[标志][域宽].[精度][类型修饰]转换格式
```

最基本的转换符由一个符号 `%` 与一个表示转换格式的字符组成，其余均为可选部分。一些常用的转换符如表 9.3 所示。

表 9.3 常用输出转换符

格式符	功能
%d 或 %i	按有符号十进制格式输出整型参数
%u	按无符号十进制格式输出无符号整型参数
%o	按无符号八进制格式输出无符号整型参数
%x	按无符号十六进制格式输出无符号整型参数, 使用字母 a, b, c, d, e, f
%X	按无符号十六进制格式输出无符号整型参数, 使用字母 A, B, C, D, E, F
%c	将整型参数转换为无符号字符型, 并输出为字符
%f	按十进制格式输出高精度浮点型参数
%e	按科学计数法格式输出高精度浮点型参数, 使用字母 e
%E	按科学计数法格式输出高精度浮点型参数, 使用字母 E
%g 或 %G	可理解为系统自动选择 %f 或 %e 格式输出
%p	按十六进制格式输出指针型参数
%s	将字符指针型参数视为字符串输出



因为格式字符串中的符号 % 有了特殊含义, 所以要原样输出一个 %, 则需要连续写两个 %, 即 %%。

使用转换符时, 可变参数中对应参数的类型应与所需的类型相匹配。因此, 所有整数类型 (字符型、短整型、整型、长整型及它们对应的无符号类型) 均被自动转换为整型或无符号整型, 所有浮点类型均被自动转换为高精度浮点类型。如果要使可变参数能够接受其他类型数据, 可以使用类型修饰。例如, 在整数类型格式前加上字母 **h** 可使之接受短整型参数, 加上字母 **l** 可使之接受长整型参数, 加上两个字母 **hh** 则可使之接受字符型参数。

域宽代表转换后数据所占的宽度, 一般是一个整数。如果转换后数据的字符数小于指定的宽度, 则在其左边补空格可以达到所需的宽度。域宽的一种特殊写法是写为符号 *****, 这时会从可变参数中取一个整数作为域宽的值。

精度与域宽之间要有一个小数点作为分隔。如果以整数格式 (如 %d, %u, %x 等) 输出, 则精度指的是最少需要输出的位数, 如果不够则在前面补 0; 如果以浮点数格式 (如 %f, %e 等) 输出, 则精度指的是小数点后要输出的数字位数; 如果以 %g 或 %G 格式输出, 则精度指的是有效数字的最多位数; 如果以 %s 格式输出, 则精度指的是要输出的字符个数。像域宽一样, 精度也可以写为字符 *****, 这时会从可变参数中取一个整数作为精度的值。

标志用于修改数据对齐和填充的方式, 常用的几个标志如表 9.4 所示。

表 9.4 常用输出格式符标志

字符	作用
数字 0	当输出数字时, 填充 0 而不是空格
减号 -	修改为左对齐方式, 空格填充在右边
空格	对于正数来说, 左边预留一个空格作为符号位
加号 +	总是在正数左边加上 + 符号, 在负数左边加上 - 符号

我们常用的 `printf` 函数实际上是对 `fprintf` 函数的包装，它用来向标准输出写入格式化的字符串，其原型如下：

```
int printf(const char *format, ...);
```

它比 `fprintf` 函数少一个文件指针参数，因为这个文件指针一定是 `stdout`。举个例子，如下的代码：

```
printf("long%*.1ld, float%12.4e\n", 8, 4, 123L, 3.1415926);
```

将在控制台上输出如下信息：

```
long    0123, float  3.1416e+00
```

与格式化输出相关的还有一个函数 `sprintf`，它并不是文件 I/O 操作，而是将格式化的字符串输出到一个缓冲区中。它的原型如下：

```
int sprintf(char *str, const char *format, ...);
```

其中 `str` 参数就指向用于存放结果的缓冲区。`sprintf` 函数会在输出字符串的末尾加上结束符 `\0`。使用这个函数时要注意，`str` 指向的缓冲区要有足够的大小来容纳生成的字符串，否则就有内存访问越界的问题。很多情况下并不能事先知道结果字符串的长度，这时可以用下面这个函数：

```
int snprintf(char *str, size_t size, const char *format, ...);
```

这个函数与 `sprintf` 函数相比多了一个参数 `size`，它的作用是限制输出字符串的长度，即写入缓冲区的字节数。如果格式化后的字符串长度等于或大于 `size`，则只写入前 `size-1` 个字节，然后写入结束符 `\0`。这里要注意，它的返回值并不是实际写入的字节数，而是格式化后的字符串的长度。

9.4.11 `fscanf`，`scanf` 和 `sscanf`

`fscanf` 可以从文件流以一定的格式读取数据，其原型如下：

```
int fscanf(FILE *stream, const char *format, ...);
```

其各个参数及返回值的含义解释如下。

- ◆ `stream`：要读取的文件指针。
- ◆ `format`：格式字符串。
- ◆ 可变参数：一般都是指针，指向用于存储读到的数据的变量。
- ◆ 返回值：成功解析的数据项的个数（不是字节数），失败则返回 `EOF`。

格式字符串中的字符将与输入流中读到的字符进行匹配，具体来说有以下几种情况。

- ◆ 空白字符：包括空格、制表、换行等字符，将与输入流中的连续 0 个或多个空白字符相匹配，也就是说，一个空白字符可以消耗多个空白字符。
- ◆ 普通字符：必须与从输入流读入的字符相同。

- ◆ 转换符：以符号 % 开始的多个字符，这时输入流中读入的字符将按某种格式解析为数据，存入对应的可变参数指向的变量中。

如果从输入流读到的字符与格式字符串不匹配，则函数将停止读操作并返回。

输入转换符的基本书写形式如下：

`%[域宽][类型修饰]转换格式`

其中方括号包围的部分为可选部分。常用的一些转换符如表 9.5 所示。

表 9.5 常用输入转换符

转换符	作用
%d	以十进制格式读入整数，存放在整型变量中
%i	当下一个字符是 0 时，以八进制格式读入整数；当下两个字符是 0x 或 0X 时，以十六进制格式读入整数；否则以十进制格式读入整数，存放在整型变量中
%u	以十进制格式读入整数，存放在无符号整型变量中
%o	以八进制格式读入整数，存放在无符号整型变量中
%x 或 %X	以十六进制格式读入整数，存放在无符号整型变量中
%f, %g, %e 或 %E	读入浮点数，存放在浮点型变量中
%s	读入字符串，字符串从下一个非空白字符开始，再遇到一个空白字符或者达到指定的域宽后结束。字符串存放在对应的参数指向的缓冲区中，末尾会自动加上 '\0'
%c	读入域宽所指定个数的字符，默认是一个。不跳过开始的空白字符，读入的字符放在对应参数指向的字符数组中，末尾不加 '\0'

可变参数中的参数类型必须与转换符对应，这可以通过类型修饰来改变。例如，在整型的转换符中加上字母 h 可使之接受短整型指针参数，加上字母 l 可使之接受长整型指针参数，加上两个字母 hh 则可使之接受字符型指针参数。

输入转换符中也可以指定域宽，它的含义是解析时读入的字符数上限。需要注意的是，多数转换符都会自动忽略开始的空白字符，这些被忽略的字符并不计算在域宽之内。

scanf 函数类似于 fscanf 函数，只不过是从标准输入读取数据。它的原型如下：

```
int scanf(const char *format, ...);
```

它比 fscanf 函数少了一个文件指针参数，因为这个指针一定是 stdin。由于用户的输入是随意的，这种格式的输入方式很难应付用户输入的各种情况，因此在一个实用的程序中很少使用 scanf 函数去接受用户输入。

还有一个 sscanf 函数可以从字符串中格式化读取数据，原型如下：

```
int sscanf(const char *str, const char *format, ...);
```

其中参数 str 就是被读取的字符串。

9.4.12 标准 I/O 错误处理

当标准 I/O 操作发生错误时，比如返回 `NULL` 指针或者 `EOF`，可以通过读 `errno` 变量得到错误码。

更方便的做法是使用标准 I/O 的错误判断函数，如：

```
int ferror(FILE *stream);  
int feof(FILE *stream);
```

`ferror` 函数用于判断文件流是否发生错误，若返回非 0 值则表示发生了错误。

`feof` 函数用于判断对文件流的读写是否已到达尾部，若返回非 0 值则表示已到达尾部。



第 10 章 深入理解进程

进程是操作系统的核心概念。本章将讨论 Linux 的进程模型，包括进程的创建、销毁及进程的状态迁移。在进程模型的基础上，本章还将探讨 Linux 信号的概念与应用。此外，依托进程的概念，本章中才得以深入讨论文件系统的运作机理。最后，本章引入进程同步的概念，并介绍进程间的通信。

10.1 Linux 中的进程

在理解进程之前，先回顾一下程序的概念。程序是一个预定义的指令序列，用来完成一个特定的任务。回忆一下所写的 C 程序，它包含各种 C 语言的语句，这些语句通常组织在函数中。一个 C 源文件可能包含变量与函数声明，类型与宏定义及各种预处理命令。一个程序的所有源文件只能且必须包含一个 `main` 函数。

依惯例，C 源文件使用 `.c` 作为扩展名，头文件使用 `.h` 作为扩展名。头文件通常只包含宏、类型定义及函数声明。头文件可以被 `#include` 预处理命令插入到源文件中。

C 编译器可以把每个源文件翻译成一个目标文件，链接器将所有的目标文件与一些必要的库链接在一起，产生一个可执行的文件。当程序被执行时，操作系统将可执行文件复制到内存中，这就是程序的映像。

进程是一个程序正在执行的实例。每个这样的实例有自己的地址空间与执行状态。进程必须有一个 PID (Process ID, 进程标识)，以便操作系统能够区分各个不同的进程。操作系统记录进程的 PID 与状态，并根据这些信息来分配系统资源（内存等）。当操作系统产生一个新的 PID，生成对应的用于管理的数据结构，并为运行程序代码分配了必要的资源，一个新的进程就产生了。可以认为进程就是一个执行的流程，在顺序执行时 CPU 的程序计数器总是指向下一条要执行的指令的地址，如果 CPU 或程序指令修改了程序计数器的内容，执行流程就发生了跳转。

现代的操作系统都能同时执行好几个进程，事实上，对于只有一个 CPU 的系统来讲，在一个给定时刻只能有一个进程在执行。操作系统内核控制 CPU 在很短的时间间隔内不断地在各个进程间切换，轮流执行。因为这个间隔很短，所以用户感觉到计算机同时在做几件事情，于是就有了“并发”执行的概念。

综上所述，一个进程具有如下核心要素。

- ◆ 程序映像：二进制指令序列。
- ◆ 地址空间：用于存放程序和执行程序。
- ◆ PCB (Process Control Block, 进程控制块)：内核中描述进程的主要数据结构。

进程管理是操作系统的核心功能。本节将介绍内核对进程的管理方式，以使读者更深入地理解

Linux 中的进程。

10.1.1 创建进程

Linux 系统上的进程间有父子关系。一个进程有且仅有一个父进程，但是可能有多个子进程。所有的进程都有一个共同的祖先，即 `init` 进程。`init` 是系统启动后创建的第一个用户态进程，它的 PID 为 1。`init` 进程对保持系统的正常运行十分重要，它会持续存在直到系统关闭，而且即使是超级用户也不能够通过信号使其终止。

Linux 系统中可以使用三个系统调用创建进程：`fork`、`vfork` 和 `clone`。前两个是所有类 UNIX 系统都提供的传统的创建进程的系统调用，而 `clone` 是 Linux 系统独有的用于创建线程的系统调用方式，它可以用来创建进程或者线程。从可移植性的角度考虑，我们不鼓励直接使用 `clone` 系统调用，如果要创建线程，可以使用 POSIX 的线程 API。

`fork` 系统调用是创建进程最常用的方式，其接口头文件与函数原型如下：

```
#include <unistd.h>
pid_t fork(void);
```

当 `fork` 调用成功返回时，系统中将会出现一个新的进程。新的进程称为原进程的子进程，而原进程则是新进程的父进程。子进程几乎完全克隆了父进程的一切特征，包括虚拟地址空间和执行进度。`fork` 函数返回一个 `pid_t` 型的进程 ID，从程序员的角度看，父子进程的唯一差异在于 `fork` 函数的返回值是不同的：父进程中返回非零值，是其子进程的进程 ID，如果是 -1，就表示创建进程失败；而在子进程中永远返回 0。这就是在程序中判断是父进程还是子进程的依据。

创建进程的另外一个系统调用是 `vfork`，其接口头文件与函数原型如下：

```
#include <sys/types.h>
#include <unistd.h>
pid_t vfork(void);
```

`vfork` 系统调用与 `fork` 基本是一样的，但是用它创建了子进程后并不完全复制父进程的虚拟地址空间。在早期的操作系统上，`fork` 系统调用会为新创建的进程分配新的物理内存以容纳从父进程复制过来的虚拟地址空间。考虑到很多时候，子进程将会立刻调用 `exec` 函数装载新的程序执行，这使得之前的内存分配与线性地址空间复制毫无意义并且浪费资源，`vfork` 的思想就是消除这个新的物理内存分配与虚拟地址空间复制的过程，让新进程的创建更有效率。使用 `vfork` 时，父进程会一直阻塞，直到子进程退出或调用 `exec` 执行新程序，并且在子进程中最好不要修改任何全局变量，因为实际上操作系统并没有为这些变量分配物理内存。

在现代的操作系统中，`fork` 调用都使用了所谓的“写时复制”(copy on write)技术，因此 `fork` 系统调用与 `vfork` 的工作效率几乎是一样的。“写时复制”的实现如下。

- ◆ 当一个进程创建时，它与父进程尽可能地共享同样的物理内存，内核仅仅复制进程的页表项并标明页的属性是“写时复制”。
- ◆ 当进程去修改内存时，就会引发一个页异常，在异常的处理过程中，内核会分配新的物理页并复制要修改的内存，然后重新进行内存映射。
- ◆ 当异常处理完毕返回后，进程修改的已经是新的物理内存，不会影响到原来与之共享内

存的其他进程。

10.1.2 执行程序

在 Linux 系统中有一系列的函数可以将一个进程的执行流程从一个可执行程序转移到另一个可执行程序，也就是装载并运行一个程序。这些函数通常被称为 `exec` 函数族，它们的接口头文件和原型如下：

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execl_e(const char *path, const char *arg, ..., char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

在解释这些函数的用法之前，我们先来了解一下变量 `environ`，它是一个每个程序都可以访问的全局变量，但访问前必须先进行声明：

```
extern char **environ;
```

实际上它是一个字符串数组的首地址，代表当前进程执行的环境变量。

`execl`、`execv` 和 `execl_e` 函数会执行由 `path` 参数指定的可执行文件。`execl` 和 `execl_e` 执行新程序时的命令行参数由参数 `arg` 及随后的可变个数参数给出，而 `execv` 函数执行新程序时的命令行参数由字符串数组参数 `argv` 给出。使用 `execl` 和 `execv` 函数时，执行新程序的环境变量取自 `environ`，即与当前进程在相同环境中执行，而使用 `execl_e` 函数时，执行新程序的环境变量由参数 `envp` 给出。

`execlp` 函数与 `execl` 函数类似，`execvp` 函数与 `execv` 函数类似，不同点在于：由 `file` 参数给出的可执行文件可以不是完整路径，系统会像 Shell 那样在 `PATH` 环境变量指定的目录中查找所要执行的文件。

对上面几个函数的总结对比如表 10.1 所示。

表 10.1 `exec` 函数族

函数	可执行文件	参数形式	环境变量
<code>execl</code>	给出全路径	可变参数列表	不提供，取自 <code>environ</code> 变量
<code>execlp</code>	在 <code>PATH</code> 环境变量中查找	可变参数列表	不提供，取自 <code>environ</code> 变量
<code>execl_e</code>	给出全路径	可变参数列表	要提供
<code>execv</code>	给出全路径	字符串数组	不提供，取自 <code>environ</code> 变量
<code>execvp</code>	在 <code>PATH</code> 环境变量中查找	字符串数组	不提供，取自 <code>environ</code> 变量

以上的函数实际上都是利用下面这个系统调用来实现的：

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

其各个参数及返回值的含义解释如下。

- ◆ filename: 要执行的程序文件。
- ◆ argv: 以 NULL 结尾的字符串数组, 表示命令行参数。
- ◆ envp: 以 NULL 结尾的字符串数组, 表示环境变量。
- ◆ 返回值: -1 表示执行失败, 如果执行成功, 则这个系统调用不会返回。

理解 `execve` 函数的关键在于当它执行成功时是不会返回的, 因为执行流程已经进入了一个新的程序。

在 `execve` 系统调用中, 内核首先会查看文件的权限。进程的所有者必须有执行这个文件的权限。如果测试失败, `execve` 函数会返回 -1, 并且将变量 `errno` 设置为 `EPERM`。通过权限检测后, 内核就会去查看文件内容, 检查程序是否真的是一个可执行文件。一般来说, Linux 系统中的可执行文件分为两类: 可执行目标文件与可执行脚本。

一、可执行目标文件

经链接器链接后可直接执行的文件称为可执行目标文件。内核一般支持几种特定格式的可执行文件。ELF 格式是 Linux 系统中普遍使用的一种标准的可执行文件格式。

ELF 格式的文件在开头有四个字节的标签, 以 `0x7f` 开始, 随后是字符 E, L, F。内核据此来判断一个文件是否是 ELF 格式的文件。并非所有的 ELF 文件都是可执行的。编译过程中产生的目标文件也是 ELF 格式的, 但没有经过最终的链接就不是可执行文件。当内核确认一个文件是 ELF 格式的文件后, 就会检查 ELF 文件头, 以确定文件是否真正可执行以及获取执行时所需的各种信息, 然后将程序加载并执行。

二、可执行脚本

可执行脚本是一个特殊的文本文件, 它能够指示内核启动一个解释器去解释执行后续的内容。这个解释器必须是可执行目标文件。如果没有合适的解释器, `execve` 将返回 -1 并且 `errno` 变量的值被设置为 `ENOEXEC`。

一般情况下, 脚本的解释器是 Shell, 但内核也会查看脚本文件第一行, 如果前两个字符是 `#!`, 它就会将第一行的剩余部分解析为启动解释器的命令。比如一个 Shell 脚本的第一行通常如下:

```
#!/bin/sh
```

这样内核将会启动 `/bin/sh` 作为脚本的解释器。

10.1.3 进程的内存布局

进程可以认为是程序运行时的一个实例, 程序中所使用的各种变量和内存存在进程的虚拟地址空间中是有一定的分布规律的, 如下面的例程:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int z = 0; /* 全局变量在数据段中 */

/* 函数在代码段中 */
int main()
```

```

{
    int *a = 0; /* 非静态的局部变量在用户栈中 */
    pid_t pid;
    if ((pid = fork())) {
        /* 父进程执行这里的代码 */
        a = (int *)malloc(100*sizeof(int)); /* 所分配的内存存在父进程的堆中 */
        z = pid;
        printf("z1 = %d\n", z);
    } else {
        /* 子进程执行这里的代码 */
        a = &z;
        *a = pid;
        printf("z2 = %d\n", z);
    }
    printf("pid = %d\n", pid);
    return 0;
}

```

在例程中通过 `fork` 函数产生了一个子进程，注意它和父进程的虚拟地址空间是相互独立的，故对全局变量的访问互不影响。程序中定义的全局变量和静态变量在运行时将放在数据段中，而非静态的局部变量则放在用户栈上，通过 `malloc` 等函数分配得到的内存则位于进程的堆中。如图 10.1 所示是一个进程的虚拟地址空间的布局。

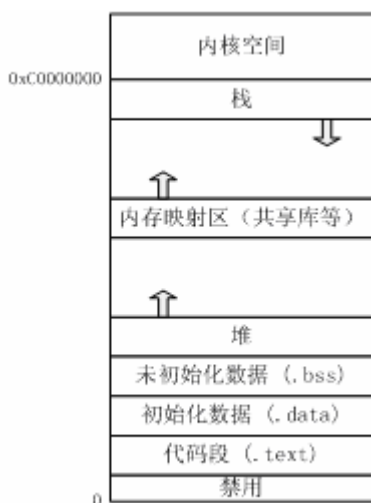


图 10.1 进程的内存布局

程序包含代码与数据，在运行前，它们要被装载入内存，这就是程序的内存映像，也可以认为是进程的内存布局，程序文件中的代码部分成为代码段，包含 CPU 要执行的指令序列；程序文件的数据部分成为数据段，包含各种全局变量和静态变量。栈是实现函数调用的基础，由内核进行分配，程序中的非静态局部变量将在栈中动态创建。堆是进行动态内存分配的场所，由内核根据需要进行分配。当创建一个进程时，子进程完全复制了父进程的代码、数据、堆和栈等。

10.1.4 进程的状态迁移

进程被创建后就处于可运行的状态，称为就绪态，但并不意味着 CPU 已经在执行这个进程，只有当内核的调度器选择了这个进程并让它在某个 CPU 上执行时，它才真正被执行，这时称进程处于运行态。内核也可以让处于运行态的进程失去 CPU，重新回到就绪态。此外，进程还有一种状态为睡眠态，在这种状态下，进程不仅停止执行，而且在内核的调度器选择下一个进入运行态的进程时不会被考虑。进程的三种状态之间的迁移可简单地理解为如图 10.2 所示。

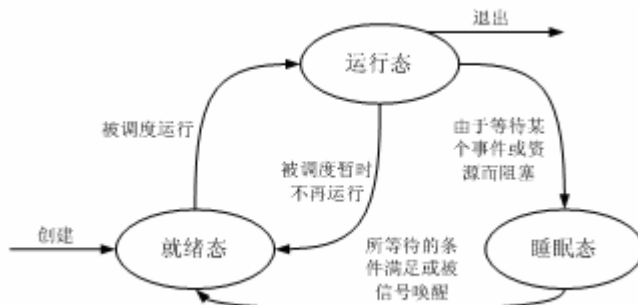


图 10.2 进程的状态转移

显然，在任何时候，处于运行态的进程数目不可能超过系统中 CPU 的数目。如果调度器认为进程已经运行了足够长的时间，并且有其他进程处于就绪态，则有可能让当前进程重新回到就绪态，而让其他进程成为运行态。

以上进程的三种状态只是理论上的概念，在 Linux 系统上实现时，由于需要支持信号、调试、跟踪等功能，进程的状态进行了更细的划分，但仍然可以用上述三种状态来概括。

10.1.5 进程的终止

一个进程在处于运行态时，可以执行退出过程而终止。从应用程序的角度来看，进程的终止方式有两种：自愿终止和被迫终止。

自愿终止指的是应用程序中主动调用了执行退出过程的系统调用而终止，这可以通过函数 `exit` 来做到，其接口头文件与原型如下：

```
#include <stdlib.h>
void exit(int status);
```

其中 `status` 参数可以指定进程退出以后的返回值。这个函数不会返回，因为调用以后进程已退出。

自愿终止也包含了从 `main` 函数返回而引起的终止，因为在这种情况下系统实际上自动调用了退出进程的系统调用。

被迫终止指的是应用程序中没有主动调用退出进程的系统调用而被内核强制终止的情形。被迫终止的位置是应用程序中无法预测的，通常是由于收到一个引起终止的信号而导致的，而引起信号的原因则是多种多样的，比如用户通过 `kill` 命令发送信号或者进程的执行发生异常等。

当一个进程终止时，内核会通知其父进程。在父进程进行处理前，这个进程成为所谓的僵尸进程，它所占用的各种资源（如内存等）已经被回收，但进程描述符仍然存在，以便父进程获取它的

退出状态。父进程可以使用 `wait` 函数或 `waitpid` 函数获取子进程的退出状态，它们的接口头文件及原型如下：

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

调用 `wait` 函数将使当前进程阻塞直到它的某个子进程退出，这时返回值是退出的子进程的 PID，而参数 `status` 指向的整数被设为子进程的退出状态（或称返回值）。如果有错误发生则返回 -1。

`waitpid` 函数则提供了比 `wait` 函数更为精细的控制，其各个参数及返回值的含义解释如下。

- ◆ `pid`：指定要等待的子进程的 PID。
- ◆ `status`：用于获取子进程的退出状态。
- ◆ `options`：等待的参数，可以改变函数的行为。
- ◆ 返回值：退出的子进程的 PID，返回 -1 表示有错误发生。

其中 `pid` 参数有如下多种用法。

- ◆ 当 `pid > 0` 时，表示某个特定的子进程。
- ◆ 当 `pid = 0` 时，表示任何进程组 ID 等于当前进程的子进程。
- ◆ 当 `pid = -1` 时，表示任何子进程。
- ◆ 当 `pid < -1` 时，表示任何进程组 ID 等于 `-pid` 的子进程。

`options` 参数的一个常用值是 `WNOHANG`，它表示函数以非阻塞的方式执行，即如果没有子进程退出则对函数的调用立刻以错误状态返回。

综上所述，对 `wait` 函数的调用 `wait(&status)` 实际上等价于：

```
waitpid(-1, &status, 0);
```

如果一个进程没有子进程，则对上述函数的调用会立刻以错误状态返回，并且变量 `errno` 的值被设为 `ECHILD`。

如果父进程在子进程退出前就已经退出，这时称子进程成为孤儿进程。在进程退出时，内核会检视它的子进程并使这些子进程成为 1 号进程的子进程，也就是说，子进程被 `init` 进程“收养”。

10.2 进程与信号

信号是内核提供了一种异步消息机制，主要用于内核对进程发送异步通知事件，可以理解为对进程的执行流程的一个“软中断”。

信号总是由内核递交给进程，但从应用程序的角度来讲，信号的来源是多种多样的，如以下所列。

- ◆ 当进程在一个没有打开的管道上等待时，内核发出 `SIGPIPE` 信号。
- ◆ 进程在 `Shell` 中前台执行时，用户按下 `Ctrl+C` 组合键，将向进程发送 `SIGINT` 信号。

- ◆ 用户使用 `kill` 命令向某个进程发送信号。
- ◆ 进程访问非法的内存地址时内核向其发送“段错误”信号 `SIGSEGV`。
- ◆ 一个进程使用系统调用向另一个进程发送信号。
- ◆ 发生各种运行时异常（如浮点错误）时，内核将向进程发送 `SIGFPE` 信号。

本节将简要介绍 Linux 系统中信号的处理机制，然后说明如何在程序中使用信号编程。

10.2.1 Linux 中的信号处理机制

在内核对进程进行管理的 PCB 信息块中有若干字节，其中每个比特位用于表示某个信号是否发生。当需要向某个进程发送一个特定的信号时，就将其 PCB 信息中对应的比特位置为 1。但是对信号的处理并不立刻发生，内核会在从内核态返回用户态时对当前进程的 PCB 中表示信号的数据进行检查，如果有信号发生，则内核会修改当前进程栈中的信息，使得返回用户态后首先执行与信号绑定的处理函数，然后再从当前进程被中断或进行系统调用的地方继续执行。如图 10.3 所示是对 Linux 系统中的信号处理机制的描述。

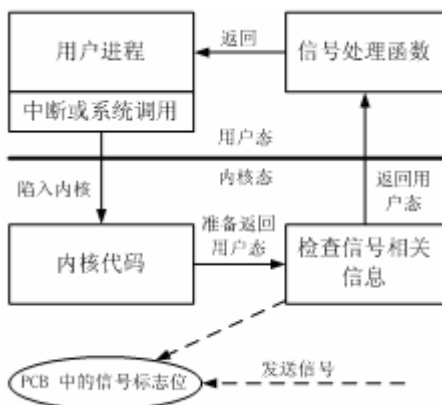


图 10.3 Linux 系统中的信号处理机制

根据以上机制，如果系统中一直没有发生从内核态返回用户态的情况，则信号就得不到处理。实际上，不光是应用程序进行系统调用时会进入内核态，当中断发生时系统也会进入内核态，因此必然会有返回用户态的时候，而且中断的发生是应用程序无法控制的，也就是说，信号何时被处理也是应用程序无法预知的。注意，信号处理函数虽然不在进程的正常执行流程中，但也是在用户态执行的代码，处于进程的上下文中，因此可以访问进程的虚拟地址空间。

下面总结一下关于信号的一些概念。

一、发送信号

给某个进程发送信号实质上就是将其 PCB 中的对应比特位置为 1。当然，这个操作只能由内核来进行，但应用程序可以通过系统调用间接地完成这个操作。

二、信号屏蔽

每个进程都有一个用来描述哪些信号将被屏蔽的信号集，称为信号掩码，如果某个信号在进程的信号掩码中，则发送到进程的这种信号将会被屏蔽。

屏蔽只是延迟信号的到达，信号会在解除屏蔽后继续传递。在信号处理函数和程序的其他部分

共享全局变量时，一般会在访问变量前屏蔽信号，以免在访问过程中发生信号而导致同步问题，访问完毕后再解除对信号的屏蔽。

三、忽略信号

忽略信号就是指当进程收到某个信号时直接丢弃，对进程的执行没有影响。但有些信号是不能被忽略的，如 **SIGKILL** 和 **SIGSTOP** 信号。

四、捕捉信号

对每个信号来说，系统都有默认的信号处理函数。如果程序中用自己定义的函数取代了默认的信号处理函数，则称为捕捉了这个信号。

10.2.2 发送信号

信号实际上是由一个正整数代表的，但为了使用方便，每个信号都定义了一个名称，代表一种特殊的含义。这些信号的定义在系统头文件 **signal.h** 中。如表 10.2 所示是 Linux 系统所支持的信号及这些信号的含义和默认操作。

表 10.2 Linux 系统中的信号

信号名	含义	默认操作
SIGABRT	进程异常终止（由 abort 函数产生）	终止进程并导出内核数据
SIGALRM	超时（由 alarm 函数产生）	终止进程
SIGFPE	浮点运算异常（被 0 除、浮点溢出等）	终止进程并导出内核数据
SIGHUP	终端挂起或控制进程终止	终止进程
SIGILL	非法硬件指令	终止进程并导出内核数据
SIGINT	终止（由用户输入 Ctrl+C 组合键产生）	终止进程
SIGKILL	终止（不能被捕捉或忽略）	终止进程
SIGPIPE	管道破裂	终止进程
SIGQUIT	终端退出符（由用户输入 Ctrl+\ 组合键产生）	终止进程并导出内核数据
SIGTERM	终止（由 kill 命令发出的默认终止信号）	终止进程
SIGUSR1	用户定义信号 1	终止进程
SIGUSR2	用户定义信号 2	终止进程
SIGSEGV	无效存储器访问（段错误）	终止进程并导出内核数据
SIGCHLD	子进程停止或退出	忽略
SIGCONT	继续执行	继续执行
SIGSTOP	停止（不能被捕捉或忽略）	暂停执行
SIGTSTP	终端挂起符（由用户输入 Ctrl+Z 组合键产生）	暂停执行
SIGTTIN	后台进程请求从终端读	暂停执行
SIGTTOU	后台进程请求向终端写	暂停执行

向一个进程发送信号可以使用 **kill** 命令，默认情况下 **kill** 命令发送 **SIGTERM** 信号以试图终止一个进程。如果要在程序中发送信号，则可以使用 **kill** 函数，其接口头文件及原型如下：

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

其中 `pid` 参数用来指定信号发送的目标进程，有以下几种情况。

- ◆ 当 `pid > 0` 时，表示目标进程的 PID。
- ◆ 当 `pid = 0` 时，表示与当前进程同组的所有进程。
- ◆ 当 `pid = -1` 时，表示所有当前进程有发送信号权限的其他进程。
- ◆ 当 `pid < -1` 时，表示所有处于进程组 `-pid` 中的进程。

`sig` 参数则是要发送的信号。函数返回 0 则表示信号发送成功，如果信号发送失败，则函数返回 -1。

需要注意的是，发送信号的进程需要有必要的权限，一般情况下，发送信号的进程与接收信号的进程应属于相同的用户。当然，超级用户可以发送信号到任何进程。

进程也可以使用 `kill` 函数给自己发送信号，但在这种情况下使用 `raise` 函数更方便，其接口头文件及原型如下：

```
#include <signal.h>
int raise(int sig);
```

与 `kill` 函数相比，它少了一个参数 `pid`，因为不需要再指明接收信号的进程。

使用 `alarm` 函数可以在过一段指定的时间后向进程自己发送 `SIGALRM` 信号，其接口头文件及原型如下：

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

其中 `seconds` 参数表示一个以秒为单位的超时时间。超过这段时间后，内核将自动向进程自己发送 `SIGALRM` 信号。如果在指定时间超时前再次调用 `alarm` 函数，则新的时间值将覆盖旧的，并且从当前时间开始重新计算。如果指定时间值为 0，表示取消信号的发送。

`alarm` 函数的返回值表示前一次对 `alarm` 函数的调用还剩多少秒就要超时，如果返回 0 则说明没有调用过 `alarm` 函数或前一次调用 `alarm` 函数后已超时。

利用 `alarm` 函数可以实现定时操作，如下面的例程：

```
#include <unistd.h>

int main()
{
    alarm(1);
    for (; ;); /* 无限循环 */
}
```

在这个例程中，由于使用 `alarm` 函数设置了 1 秒后发送 `SIGALRM` 信号，而 `SIGALRM` 信号的默认操作是退出进程，所以尽管随后是一个无限循环，进程仍然能够退出。

10.2.3 捕捉信号

10.2.3.1 使用 signal 函数

signal 函数是 Linux 系统上传统的信号处理接口，其接口头文件及原型如下：

```
#include <signal.h>
sighandler_t signal(int signum, sighandler_t handler);
```

其中的 sighandler_t 类型是一个函数指针类型，定义如下：

```
typedef void (*sighandler_t)(int);
```

这个类型表示一个信号处理函数。signal 函数的作用就是将 handler 参数所指向的函数注册成为参数 signum 所代表的信号的处理函数，它的返回值是这个信号原来的处理函数，如果返回 SIG_ERR，则说明有错误发生，注册失败。

注册成功以后，所注册的函数就会在信号被处理时调用，代替了默认的行为，或者称为信号被捕捉。

使用 signal 函数时应注意如下两点。

- ◆ handler 参数的值可以是 SIG_IGN 或者 SIG_DFL, SIG_IGN 表示忽略这个信号, SIG_DFL 表示对信号的处理重设为系统的默认方式。
- ◆ 有些信号是不可以忽略或捕获的，如 SIGKILL 和 SIGSTOP。

下面给出一个例程来说明信号的产生、忽略与捕获的编程，例程代码如下：

```
/* 文件名: sigtest.c */
/* 说明: 信号处理例程 */

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>

static pid_t pid;

/* 子进程 1 SIGALRM 信号处理函数 */
static void wakeup(int dummy)
{
    printf("I (pid = %d) am up now\n", pid);
}

/* 子进程 1 SIGINT 信号处理函数 */
static void handler(int dummy)
{
    printf("I (pid = %d) got an interrupt, will exit\n", pid);
    exit(0);
}
```



```
/* 子进程 2 信号处理函数 */
static void trapper(int i)
{
    if (i == SIGUSR1) {
        printf("I (pid = %d) got SIGUSR1, will exit\n", pid);
        exit(0);
    } else {
        printf("I (pid = %d) got signal %d, will continue\n", pid, i);
    }
}

/* 父进程信号处理函数 */
void parent(int sig)
{
    printf("Signal (%d) received by parent (%d)\n", sig, pid);
}

int main(int argc, char *argv[])
{
    int i, cpid1, cpid2;
    printf("Number of signal is %d\n", NSIG); /* 输出系统中信号的个数 */
    if (!(cpid1 = fork())) { /* 创建第一个子进程 */
        pid = cpid1 = getpid(); /* 取得子进程的进程号 */
        printf("CPID1 = %d\n", cpid1);
        for (i = 1; i < NSIG; i++) signal(i, SIG_IGN); /* 忽略所有信号 */
        signal(SIGINT, handler); /* 捕获信号 SIGINT */
        signal(SIGALRM, wakeup); /* 捕获超时信号 */
        alarm(2); /* 启动定时器, 设置 2 秒后超时 */
        for (;;) {
            pause(); /* 等待信号 */
        }
        printf(" -- CPID1 (%d) terminates\n", cpid1);
        exit(0);
    } else if (!(cpid2 = fork())) { /* 创建第二个子进程 */
        pid = cpid2 = getpid();
        printf("CPID2 = %d\n", cpid2);
        for (i = 1; i < NSIG; i++) {
            signal(i, trapper); /* 捕获所有的信号 */
        }
        for (;;) {
            pause(); /* 等待信号 */
        }
        printf(" -- CPID2 (%d) terminates\n", cpid2);
        exit(0);
    }
    /* 下面是父进程执行的代码 */
    pid = getpid(); /* 取得 PID */
    sleep(3); /* 睡眠, 让子进程先运行 */
    printf("This is parent process (pid = %d)\n", pid);
    for (i = 1; i < NSIG; i++) {
```



```

    signal(i, parent); /* 捕获所有的信号 */
}
printf("Send SIGUSR1(%d) to CPID1 (%d)\n", SIGUSR1, cpid1);
kill(cpid1, SIGUSR1);
printf("Send SIGINT(%d) to CPID1 (%d)\n", SIGINT, cpid1);
kill(cpid1, SIGINT);
printf("Send SIGINT(%d) to CPID2 (%d)\n", SIGBUS, cpid2);
kill(cpid2, SIGINT);
printf("Send SIGUSR1(%d) to CPID2 (%d)\n", SIGUSR1, cpid2);
kill(cpid2, SIGUSR1);
for (; wait((int *)0) > 0; ); /* 等待子进程结束 */
return 0;
}

```

在这个例程中，父进程又创建了两个子进程，这三个进程分别注册或忽略了相关的信号，然后通过 `alarm` 函数设置超时信号或通过 `kill` 函数发送信号。在例程中还用到了 `pause` 和 `sleep` 函数，它们的接口头文件及原型如下：

```

#include <unistd.h>
int pause(void);
unsigned int sleep(unsigned int seconds);

```

`pause` 函数将使当前进程进入睡眠态，直到有信号发生。它的返回值永远是 `-1`，同时变量 `errno` 的值被设为 `EINTR` 以表示有信号发生。

`sleep` 函数将使当前进程进入睡眠态，并在 `seconds` 参数指定的秒数后被唤醒继续执行。注意当有未忽略的信号发生时 `sleep` 函数会提前返回，返回值是剩余的秒数。如果返回 `0` 则说明是正常被唤醒的，而不是因信号的发生提前返回的。

10.2.3.2 使用 `sigaction` 函数

`signal` 函数的使用方法比较简单，但并不属于 POSIX 标准，在各种类 UNIX 平台上的实现不尽相同，因此其用途受到了一定的限制。而 POSIX 标准定义的信号处理接口是 `sigaction` 函数，其接口头文件及原型如下：

```

#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);

```

其各个参数及返回值的含义解释如下。

- ◆ `signum`：要操作的信号。
- ◆ `act`：要设置的对信号的新处理方式。
- ◆ `oldact`：原来对信号的处理方式。
- ◆ 返回值：`0` 表示成功，`-1` 表示有错误发生。

`struct sigaction` 类型用来描述对信号的处理，定义如下：

```

struct sigaction {

```



```

void      (*sa_handler)(int);
void      (*sa_sigaction)(int, siginfo_t *, void *);
sigset_t   sa_mask;
int        sa_flags;
void      (*sa_restorer)(void);
};

```

在这个结构体中, 成员 `sa_handler` 是一个函数指针, 其含义与 `signal` 函数中的信号处理函数类似。成员 `sa_sigaction` 则是另外一个信号处理函数, 它有三个参数, 可以获得关于信号的更详细的信息。当 `sa_flags` 成员的值包含了 `SA_SIGINFO` 标志时, 系统将使用 `sa_sigaction` 函数作为信号处理函数, 否则使用 `sa_handler` 作为信号处理函数。在某些系统中, 成员 `sa_handler` 与 `sa_sigaction` 被放在联合体中, 因此使用时不要同时设置。

`sa_mask` 成员用来指定在信号处理函数执行期间需要被屏蔽的信号, 特别是当某个信号被处理时, 它自身会被自动放入进程的信号掩码, 因此在信号处理函数执行期间这个信号不会再度发生。

`sa_flags` 成员用于指定信号处理的行为, 它可以是以下值的“按位或”组合。

- ◆ `SA_RESTART`: 使被信号打断的系统调用自动重新发起。
- ◆ `SA_NOCLDSTOP`: 使父进程在它的子进程暂停或继续运行时不会收到 `SIGCHLD` 信号。
- ◆ `SA_NOCLDWAIT`: 使父进程在它的子进程退出时不会收到 `SIGCHLD` 信号, 这时子进程如果退出也不会成为僵尸进程。
- ◆ `SA_NODEFER`: 使对信号的屏蔽无效, 即在信号处理函数执行期间仍能发生这个信号。
- ◆ `SA_RESETHAND`: 信号处理之后重新设置为默认的处理方式。
- ◆ `SA_SIGINFO`: 使用 `sa_sigaction` 成员而不是 `sa_handler` 作为信号处理函数。

`sa_restorer` 成员则是一个已经废弃的数据域, 不要使用。

下面用一个例程来说明 `sigaction` 函数的使用, 代码如下:

```

/* 文件名: sigaction.c */
/* 说明: 信号处理例程 */

#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

static void sig_usr(int signum)
{
    if(signum == SIGUSR1) {
        printf("SIGUSR1 received\n");
    } else if (signum == SIGUSR2) {
        printf("SIGUSR2 received\n");
    } else {
        printf("signal %d received\n", signum);
    }
}

int main(void)

```



```

{
    char buf[512];
    int n;
    struct sigaction sa_usr;
    sa_usr.sa_flags = 0;
    sa_usr.sa_handler = sig_usr; /* 信号处理函数 */
    sigaction(SIGUSR1, &sa_usr, NULL);
    sigaction(SIGUSR2, &sa_usr, NULL);
    printf("My PID is %d\n", getpid());
    while (1) {
        if ((n = read(STDIN_FILENO, buf, 511)) == -1) {
            if (errno == EINTR) { /* read 函数被信号打断 */
                printf("read is interrupted by signal\n");
            }
        } else {
            buf[n] = '\0';
            printf("%d bytes read: %s\n", n, buf);
        }
    }
    return 0;
}

```

在这个例程中使用 `sigaction` 函数为 `SIGUSR1` 和 `SIGUSR2` 信号注册了处理函数，然后从标准输入读入字符。程序运行后将首先输出自己的 PID，如：

```
My PID is 5904
```

这时如果从另外一个终端向进程发送 `SIGUSR1` 或 `SIGUSR2` 信号，用类似如下的命令：

```
kill -USR1 5904
```

则程序将继续输出如下内容：

```
SIGUSR1 received
read is interrupted by signal
```

这说明用 `sigaction` 注册信号处理函数时，不会自动重新发起被信号打断的系统调用。如果要自动重新发起，则要设置 `SA_RESTART` 标志，比如在上述例程中可以进行类似以下的设置：

```
sa_usr.sa_flags = SA_RESTART;
```

10.2.4 屏蔽信号

在 `sigaction` 函数的使用中，我们已经看到了表示信号集的 `sigset_t` 型数据。在 Linux 上有一组函数专门用于对信号集进行操作，它们的接口头文件及原型如下：

```

#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);

```

```
int sigismember(const sigset_t *set, int signum);
```

其中 `set` 参数指向要操作的信号集，而 `signum` 参数则代表一个指定的信号，各个函数的作用如下。

- ◆ `sigemptyset`: 清空信号集，返回 0 表示成功，-1 表示失败。
- ◆ `sigfillset`: 将所有信号加入信号集，返回 0 表示成功，-1 表示失败。
- ◆ `sigaddset`: 将指定信号加入信号集，返回 0 表示成功，-1 表示失败。
- ◆ `sigdelset`: 将指定信号从信号集中去除，返回 0 表示成功，-1 表示失败。
- ◆ `sigismember`: 判断一个指定的信号是否在信号集中，返回 1 表示在信号集中，0 表示不在信号集中，-1 表示有错误发生。

有了这些函数，编程时就没有必要直接对 `sigset_t` 型数据进行操作了，从而也不必知道它的具体定义。一般来说，信号集在使用前需要先用 `sigemptyset` 或 `sigfillset` 函数进行初始化，然后用 `sigaddset` 或 `sigdelset` 函数增加或去除需要的信号。

在调用 `sigaction` 函数时，可以设置信号处理时要屏蔽的信号。实际上在代码中也可以直接设置或获取进程的信号掩码，所用的接口函数如下：

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

其各个参数及返回值的含义解释如下。

- ◆ `how`: 指定操作信号掩码的方式。
- ◆ `set`: 指向用于设置信号掩码的信号集。
- ◆ `oldset`: 用于返回原来的信号掩码。
- ◆ 返回值: 0 表示成功，-1 表示失败。

`sigprocmask` 函数将根据参数 `how` 指定的方式，设置当前进程的信号掩码，并把原来的信号掩码保存在参数 `oldset` 指向的信号集中返回。如果 `set` 参数为 `NULL`，则不修改信号掩码；如果 `oldset` 参数为 `NULL`，则不返回原来的信号掩码。这里关键要理解参数 `how` 的使用，它有如下三个取值。

- ◆ `SIG_BLOCK`: 将 `set` 参数指向的信号集中的信号加入到信号掩码中。
- ◆ `SIG_UNBLOCK`: 将 `set` 参数指向的信号集中的信号从信号掩码中删除。
- ◆ `SIG_SETMASK`: 将 `set` 参数指向的信号集设置为信号掩码。

因此，屏蔽某个信号可以有两种方式，下面以 `SIGUSR1` 信号为例进行说明。

第一种方式为使用 `SIG_BLOCK` 操作方式，代码示例如下：

```
sigset_t sigset;
sigemptyset(&sigset);
sigaddset(&sigset, SIGUSR1);
sigprocmask(SIG_BLOCK, &sigset, NULL);
```

第二种方式为使用 `SIG_SETMASK` 操作方式，代码示例如下：

```
sigset_t set;
sigprocmask(SIG_SETMASK, NULL, &set); /* 先得到当前的信号掩码 */
sigaddset(&set, SIGUSR1); /* 将要屏蔽的信号加入 */
sigprocmask(SIG_SETMASK, &set, NULL);
```

同样，要解除对信号的屏蔽，也有两种方式，仍以 SIGUSR1 信号为例进行说明。

第一种方式，使用 SIG_UNBLOCK 操作方式，代码示例如下：

```
sigset_t sigset;
sigemptyset(&sigset);
sigaddset(&sigset, SIGUSR1);
sigprocmask(SIG_UNBLOCK, &sigset, NULL);
```

第二种方式，使用 SIG_SETMASK 操作方式，代码示例如下：

```
sigset_t set;
sigprocmask(SIG_SETMASK, NULL, &set); /* 先得到当前的信号掩码 */
sigdelset(&set, SIGUSR1); /* 将要解除屏蔽的信号去除 */
sigprocmask(SIG_SETMASK, &set, NULL);
```

10.2.5 信号安全函数

如前所述，进程在收到信号并对其进行处理时，会中断当前正在执行的指令序列，而去执行信号处理函数。但是信号的传递是异步的，系统无法确定何时传递信号给进程。如果进程在收到信号时正在执行某个不可重入的函数，这时捕捉到信号，进程就会转而去执行信号处理函数。如果在这个信号处理函数中又再次调用了同一个函数，就有可能产生问题。

因此有些函数是不能在信号处理函数中调用的，那些可以在信号处理函数中调用且不会有潜在问题的函数称为对信号安全的，表 10.3 列出了 Linux 系统中的信号安全函数，对于不在这个表中的函数，如果要在信号处理函数中调用，则需要谨慎处理。

表 10.3 Linux 系统中的信号安全函数

_Exit()	_exit()	abort()	accept()
access()	aio_error()	aio_return()	aio_suspend()
alarm()	bind()	cfgetispeed()	cfgetspeed()
cfsetispeed()	cfsetospeed()	chdir()	chmod()
chown()	clock_gettime()	close()	connect()
creat()	dup()	dup2()	execle()
execve()	fchmod()	fchown()	fcntl()
fdatasync()	fork()	fpathconf()	fstat()
fsync()	ftruncate()	getegid()	geteuid()
getgid()	getgroups()	getpeername()	getpggrp()
getpid()	getppid()	getsockname()	getsockopt()
getuid()	kill()	link()	listen()
lseek()	lstat()	mkdir()	mkfifo()

(续表)

open()	pathconf()	pause()	pipe()
poll()	posix_trace_event()	pselect()	raise()
read()	readlink()	recv()	recvfrom()
recvmsg()	rename()	rmdir()	select()
sem_post()	send()	sendmsg()	sendto()
setgid()	setpgid()	setsid()	setsockopt()
setuid()	shutdown()	sigaction()	sigaddset()
sigdelset()	sigemptyset()	sigfillset()	sigismember()
signal()	sigpause()	sigpending()	sigprocmask()
sigqueue()	sigset()	sigsuspend()	sleep()
socketmark()	socket()	socketpair()	stat()
symlink()	sysconf()	tcdrain()	tcflow()
tcflush()	tcgetattr()	tcgetpgrp()	tcsendbreak()
tcsetattr()	tcsetpgrp()	time()	timer_getoverrun()
timer_gettime()	timer_settime()	times()	umask()
uname()	unlink()	utime()	wait()
waitpid()	write()		

10.3 进程与文件

由于 Linux 系统中文件概念的多样性与抽象性，虽然其编程接口已经设计得相当简洁，但对文件的操作仍然是比较复杂的。在嵌入式开发中，由于要对各种设备进行操作，必须深入理解文件的编程接口及其实现的原理。

本节将首先介绍内核对文件管理的基本原理，然后以几个典型应用场景来说明进程与文件的关系。

10.3.1 内核文件管理

在内核中，对应于每个进程都有一个文件描述符表，表示这个进程打开的所有文件。文件描述符表中的每一项都是一个指针，指向一个用于描述打开的文件的数据块，可称为 **file** 对象。**file** 对象中描述了文件的打开模式、读写位置等重要信息。当进程打开一个文件时，内核中就会创建一个新的 **file** 对象。需要注意，**file** 对象不是专属于某个进程的，不同进程的文件描述符表中的指针可以指向相同的 **file** 对象，从而共享这个打开的文件。**file** 对象有引用计数，记录了引用这个对象的文件描述符的个数，只有当引用计数为 0 时，内核才销毁 **file** 对象，因此某个进程关闭文件不影响与之共享同一个 **file** 对象的进程。

file 对象中包含一个指针，指向 **dentry** 对象。**dentry** 对象代表一个独立的文件路径，如果一个文件路径被打开多次，那么会建立多个 **file** 对象，但它们都指向同一个 **dentry** 对象。

dentry 对象中又包含一个指向 **inode** 对象的指针。**inode** 对象代表一个独立的文件。因为在

Linux 的文件系统中存在硬链接和符号链接, 因此不同的 **dentry** 对象可能指向相同的 **inode** 对象。**inode** 对象包含了最终对文件进行操作所需的所有信息, 如文件系统类型、文件的操作方法、文件的权限、访问日期等。

需要注意的是, 上述文件描述符表、**file** 对象、**dentry** 对象和 **inode** 对象都是内核用于文件管理的数据, 应用程序是不可能直接访问的。

如图 10.4 所示是三个进程打开同一个文件后内核产生的各种管理数据的关系, 其中进程 1 和进程 2 使用相同的路径打开了同一个文件, 进程 3 则使用了另外一个路径, 但这两个路径实际上最终指向同一个索引节点。

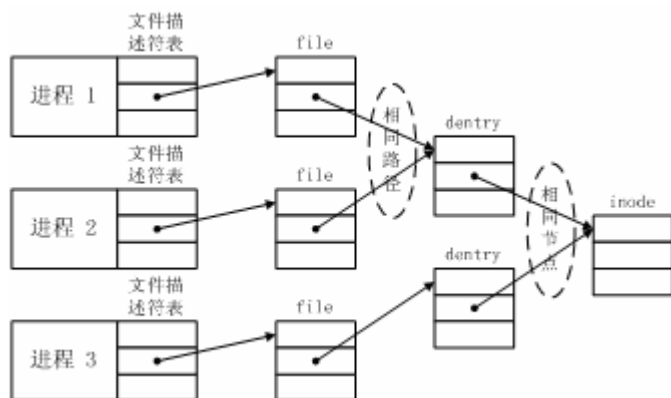


图 10.4 内核文件管理示意

打开文件后, 进程得到的文件描述符实质上就是文件描述符表的下标, 内核根据这个下标值去访问相应的文件对象, 从而实现对文件的操作。

10.3.2 进程中的文件

下面将从不同的角度讲述进程与文件的关系以及在进程中如何处理文件。

我们先来回顾一下在进程中如何对文件进行操作。在 C 语言程序中, 文件是以文件指针及文件描述符表示的。标准的 C 库函数 (**fopen**, **fscanf**, **fprintf**, **fread**, **fwrite**, **fclose** 等) 使用文件指针, 而系统的 I/O 功能 (**open**, **read**, **write**, **close**, **ioctl** 等) 使用文件描述符。实际上, 文件指针是对文件描述符的一种包装。

表示标准输入、标准输出及标准错误输出的文件指针分别是 **stdin**, **stdout**, **stderr**, 这些变量定义在头文件 **stdio.h** 中; 而相应的文件描述符分别是 **STDIN_FILENO**, **STDOUT_FILENO**, **STDERR_FILENO**, 这些宏定义在头文件 **unistd.h** 中。

使用 **open** 函数可以打开一个文件, 这时内核将创建一个新的 **file** 对象, 并在进程的文件描述符表中寻找一个空位存放对象的指针, 然后返回它的下标作为文件描述符。在随后的文件操作中, 应用程序使用文件描述符即可完全代表这个打开的文件。注意, 同一个进程多次打开同一个文件时, 内核会创建多个 **file** 对象。

当进程使用 **fork** 系统调用创建一个子进程后, 子进程将继承父进程的文件描述符表, 因此在父进程中打开的文件可以在子进程中用同一个描述符访问, 如图 10.5 所示。

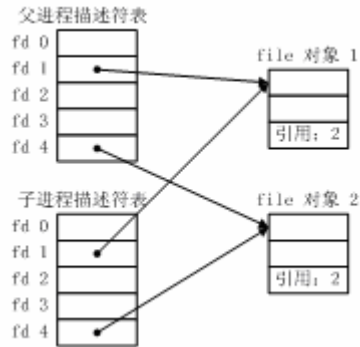


图 10.5 子进程继承父进程的文件描述符表

下面将通过一个例程来说明创建新进程及执行新程序时对已打开文件的处理。例程由两个源文件组成：**file_fork.c** 和 **openfexec.c**。

file_fork.c 文件的内容如下：

```
/* 文件名: file_fork.c */
/* 说明: 进程与文件例程 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    int fd;
    int cflag, excode;
    long pos;
    /* 打开文件 */
    fd = open("file_fork.c", O_RDONLY);
    printf("fd = %d\n", fd);
    pos = lseek(fd, 20L, SEEK_SET); /* 设置文件的读写位置到 20 */
    printf("Current position before fork() is %ld\n", pos);
    if (!fork()) { /* 创建子进程 */
        pos = lseek(fd, 40L, SEEK_SET); /* 设置文件的读写位置到 40 */
        printf("Current position in child after fork() is %ld\n", pos);
        exit(0);
    }
    /* 父进程执行的代码 */
    wait((int *)0); /* 等待子进程退出 */
    pos = lseek(fd, 0L, SEEK_CUR); /* 获取文件当前读写位置 */
    printf("Current position in parent after fork() is %ld\n", pos);
    if (!fork()) { /* 再次创建子进程 */
        execl("./openfexec", "./openfexec", (char *)0); /* 执行程序 */
        printf("It is an error to print this line out\n");
        exit(-1);
    }
}
```



```

    }
    /* 父进程执行的代码 */
    wait(&excode); /* 等待子进程退出 */
    pos = lseek(fd, 0L, SEEK_CUR);
    printf("Current pos in parent after exec() is %ld\n", pos);
    printf("Exit code of a child = %d\n", WEXITSTATUS(excode));
    cflag = fcntl(fd, F_GETFD); /* 取得 close-on-exec 标志 */
    printf("close-on-exec flag = %d\n", cflag);
    fcntl(fd, F_SETFD, FD_CLOEXEC); /* 设置 close-on-exec 标志 */
    if (!fork()) { /* 再次创建子进程 */
        execlp("./openfexec", "./openfexec", (char *)0); /* 执行程序 */
        printf("It is an error to print this line out\n");
        exit(-1);
    }
    /* 父进程执行的代码 */
    wait(&excode); /* 等待子进程退出 */
    printf("Exit code of a specific child = %d\n", WEXITSTATUS(excode));
    return 0;
}

```

在上述代码中使用 `execl` 函数执行了当前目录下的程序 `openfexec`，它的源代码如下：

```

/* 文件名: openfexec.c */
/* 说明: 进程与文件例程 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd = 3; /* 原进程中只打开过一个文件，所以描述符为 3 */
    long pos;
    pos = lseek(fd, 0L, SEEK_CUR);
    if (pos == (off_t)-1) {
        perror("openfexec");
        exit(1);
    }
    printf("Current position in openfexec is %ld\n", pos);
    pos = lseek(fd, 50L, SEEK_CUR); /* 设置文件位置到 50 */
    if (pos == (off_t)-1) {
        perror("openfexec");
        exit(2);
    }
    printf("New position in openfexec is %ld\n", pos);
    return 0;
}

```

将 `openfexec.c` 源程序编译为可执行文件 `openfexec`，然后编译 `file_fork.c` 并运行，结果如下：

```

fd = 3
Current position before fork() is 20
Current position in child after fork() is 40
Current position in parent after fork() is 40
Current position in openfexec is 40
New position in openfexec is 90
Current pos in parent after exec() is 90
Exit code of a child = 0
close-on-exec flag = 0
openfexec: Bad file descriptor
Exit code of a specific child = 1

```

从结果中可以看出，子进程与父进程共享了相同的 `file` 对象，由于文件的读写位置是保存在 `file` 对象中的，所以子进程对读写位置的修改也影响到了父进程。

需要注意的是，使用 `exec` 函数族执行新的程序时，默认情况下不会关闭原来已经打开的文件描述符，因此在新的程序中不需要打开文件就可以直接使用原来的文件描述符进行操作。这有时候是不安全的，可以设置文件的 `close-on-exec` 属性，如例程中那样，这样使用 `exec` 函数族执行新的程序时文件就会自动被关闭。

10.3.3 文件的重定向

在 `Shell` 命令中可以使用文件的重定向。实际上，`Linux` 系统提供了重定向文件描述符的系统调用，因此在编程时也可以实现文件的重定向。

我们已经知道，文件描述符就是对进程的文件描述符表进行索引的下标。文件描述符表中的每个条目指向一个 `file` 对象，因此如果能修改文件描述符表中的条目，使之指向其他 `file` 对象，就可以实现重定向。这可以通过函数 `dup` 和 `dup2` 来实现，它们的接口头文件和原型如下：

```

#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);

```

`dup` 函数的参数 `oldfd` 是一个已打开的文件描述符，这个函数可以返回一个新的文件描述符，与 `oldfd` 代表同一个已打开的文件。`dup2` 函数则提供了更多的控制，可以用参数 `newfd` 来指定新的文件描述符，相当于将 `newfd` 重定向到了 `oldfd`。这两个函数都能返回新的文件描述符，失败则返回 `-1`。

下面以一个例子来说明 `dup2` 函数的作用。假设一个进程中已经打开了两个文件，分别使用文件描述符 1 和 4，如图 10.6 所示。

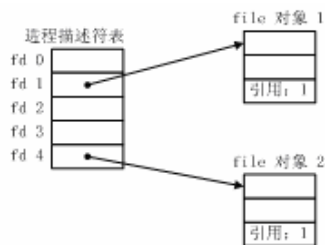


图 10.6 重定向前的文件描述符表

调用 `dup2(4, 1)` 后, 描述符 1 被重定向到 4, 它与描述符 4 指向同一个文件, 如图 10.7 所示。描述符 1 原来指向的 `file` 对象的引用计数会下降 1, 如果降到 0, 这个对象就会被内核销毁。

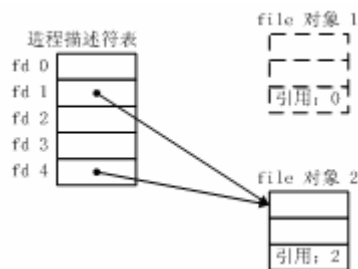


图 10.7 重定向后的文件描述符表

10.3.4 文件控制

`fcntl` 函数是一个对文件进行控制的通用函数, 它能够获取与设置一个打开的文件的控制属性, 其接口头文件与原型如下:

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

`fcntl` 函数的 `fd` 参数是一个文件描述符, 指定要操作的文件, 它的具体功能由参数 `cmd` 指定, 用于不同的功能时, 其他参数的含义也有所不同, 甚至参数的个数也可以不同。

`fcntl` 执行成功时, 返回值的含义与具体的功能有关, 执行失败时返回 `-1`。

表 10.4 中列出了 `fcntl` 函数的几个常用功能。

表 10.4 `fcntl` 函数的常用功能

cmd	arg 或 lock	功能描述	成功时返回值
<code>F_DUPFD</code>	文件描述符	寻找一个大于等于 <code>arg</code> 的空闲描述符并将其重定向到 <code>fd</code>	新的文件描述符
<code>F_GETFD</code>	无	获取文件的 <code>close-on-exec</code> 属性	<code>close-on-exec</code> 属性
<code>F_SETFD</code>	<code>FD_CLOEXEC</code>	设置文件的 <code>close-on-exec</code> 属性	0
<code>F_GETFL</code>	无	获取文件标志	文件标志
<code>F_SETFL</code>	新的标志	设置文件标志	0
<code>F_SETLK</code>	指向文件锁操作相关数据	获取或释放文件锁, 获取时如果与文件已有的锁冲突则返回失败	0
<code>F_SETLKW</code>	指向文件锁操作相关数据	获取或释放文件锁, 获取时如果与文件已有的锁冲突则进程阻塞, 等待到冲突消失 (其他进程释放了文件锁)	0
<code>F_GETLK</code>	指向文件锁操作相关数据	得到文件锁的状态	0
<code>F_GETOWN</code>	无	得到异步操作文件时相关信号的接收进程	进程的 PID
<code>F_SETOWN</code>	进程的 PID	设置异步操作文件时相关信号的接收进程	0



下面举一个例子，在文件打开之后为其增加非阻塞操作标志，代码如下：

```
/* 文件名: fcntl.c */
/* 说明: fcntl 函数应用例程 */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>

#define MSG "input again\n"

int main(void)
{
    char buf[32];
    int n;
    int flags;
    flags = fcntl(STDIN_FILENO, F_GETFL); /* 获取文件标志 */
    flags |= O_NONBLOCK; /* 增加非阻塞标志 */
    if (fcntl(STDIN_FILENO, F_SETFL, flags) == -1) { /* 设置新标志 */
        perror("fcntl");
        exit(-1);
    }
again:
    n = read(STDIN_FILENO, buf, 32);
    if (n < 0) {
        if (errno == EAGAIN) { /* 无数据可读 */
            sleep(1);
            write(STDOUT_FILENO, MSG, strlen(MSG));
            goto again;
        }
        /* 其他情况视为错误 */
        perror("read");
        exit(-1);
    }
    write(STDOUT_FILENO, buf, n); /* 输出读到的数据 */
    return 0;
}
```

在上述例程中，由于标准输入被改成非阻塞操作，故可以在监视标准输入的同时进行其他操作，如输出信息到标准输出。

10.4 进程间通信

一些复杂的应用可能会需要多个进程分工协作来满足所需的功能要求，这就必然涉及到数据在进程之间的共享或交换，称为 IPC（Inter-process communication，进程间通信）。本节将主要讲



述 Linux 系统上 IPC 接口的使用。

10.4.1 Linux 中的 IPC

由于历史的原因，在类 UNIX 系统的发展中，不同的团队发展出了不同的解决方案。这些方案被 Linux 系统兼收并蓄，并经过自己的改进，形成了支持最广泛的解决方案。各种解决方案的发展历程如图 10.8 所示。

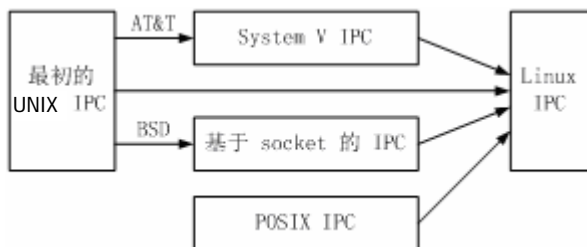


图 10.8 IPC 方案发展历程

Linux 系统的 IPC 接口主要由 System V IPC、POSIX IPC 及 BSD Socket 构成。其中 BSD Socket 就是基于 socket 的方式。socket 不仅可以用于同一主机上的各个进程之间通信，更主要的是可以用于不同主机间的网络通信。

System V IPC 接口是由 AT&T 的贝尔实验室发展出来的，其通信机制主要包括管道、FIFO、消息队列、信号灯、共享内存等。

由于 UNIX 的分支版本众多，为提高应用程序的可移植性，IEEE（The Institute of Electrical and Electronics Engineers，电气和电子工程师协会）制定了 POSIX（Portable Operating System Interface，可移植操作系统接口）。POSIX 标准的内容十分广泛，包括 POSIX IPC 接口。POSIX IPC 主要包括消息队列、信号灯、共享内存等。

现在的大部分类 UNIX 操作系统都支持 POSIX 标准。Linux 系统从一开始就遵循 POSIX 标准，因此一般情况下推荐使用 POSIX IPC 接口。

下面对 Linux 系统中进程间通信的几种主要方式做简单的说明。

一、管道和命名管道

管道可用于具有亲缘关系的进程间的通信。命名管道克服了管道没有名字的限制，因此可以用于无亲缘关系的进程间的通信。所有的 Linux/UNIX 系统平台都支持管道及命名管道，因此有良好的可移植性。

二、消息队列

消息队列是消息的链表，有足够权限的进程可以向队列中添加消息，有读队列权限的进程则可以读走队列中的消息。消息队列支持消息类型与优先级。Linux 系统支持两种消息队列：POSIX 消息队列和 System V 消息队列。

三、共享内存

使用共享内存可以让多个进程访问同一块内存空间，是效率最高的 IPC 方式，但可能会有同步问题。Linux 系统支持两种共享内存机制：POSIX 共享内存和 System V 共享内存。

四、信号灯

信号灯主要用于进程间或线程间的同步。Linux 系统支持两种信号灯：POSIX 信号灯和 System V 信号灯。

五、socket

socket 可用于本地进程间的通信，也可用于网络通信，是各种操作系统普遍支持的一种通信方式，因此可移植性也比较好。

本节的以下部分将介绍各种 IPC 接口的使用，主要是 POSIX IPC。

10.4.2 信号灯与进程同步

多个进程间共享或交换数据时要考虑的一个重要方面就是进程间可能需要进行同步。因此，在讨论具体的通信方式之前，最好先搞清楚进程同步的概念。

10.4.2.1 同步问题的产生

Linux 是多任务的操作系统，允许多个进程的并发执行。当多个进程同时对一个系统资源（如同一块内存）进行访问时，有可能产生一些不确定的问题。下面通过一个例程来看一下 Linux 系统中的同步问题。假定需要有两个进程协作向控制台输出一句完整的话，并要求其中一个进程输出上半句，而另一个进程输出下半句，并且输出每个字符后要停顿 1 秒钟再输出下一个字符。例程代码如下：

```
/* 文件名: mangle.c */
/* 说明: 进程同步问题 */

#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int i, j;
    pid_t pid;
    const char *message = "Hello World!\n";
    int n = strlen(message) / 2;
    pid = fork(); /* 生成新进程 */
    j = (pid == 0) ? 0 : n; /* 子进程从头开始输出，父进程从中间开始输出 */
    for (i = 0; i < n; i++) {
        write(STDOUT_FILENO, message + j + i, 1);
        sleep(1); /* 睡眠，这样就会调度到其他进程 */
    }
    return 0;
}
```

编译运行程序，结果是一堆杂乱的输出。主要的问题是两个进程运行的顺序是不确定的，由内核的调度器来决定。

为了解决进程同步问题，需要一种机制来协调多个进程的执行过程。具体在这个例子中，如果

能让负责输出上半句的进程先输出完毕后,再让负责输出下半句的进程开始输出,问题则得到解决。

10.4.2.2 使用信号灯解决同步问题

在 Linux 系统中可以使用信号灯来解决上述的同步问题。从概念上讲,信号灯有一个计数值,用于表示可用资源的数量。对信号灯的基本操作包括获取与释放。获取信号灯时,如果它的值已经是 0 则不能被获取,进程将阻塞直到它的值大于 0,然后将它的值减 1;如果它的值大于 0,则直接将它的值减 1。释放信号灯就是将信号灯的值得加 1。

POSIX 标准中的信号灯分为两类:命名信号灯和无名信号灯。命名信号灯有一个全局唯一的名字,因而可以在两个没有亲缘关系的进程间使用。无名信号灯是基于内存变量的一种信号灯,如果要在两个进程间使用则必须把它放在两者的共享内存中。

关于信号灯的接口头文件和函数原型如下:

```
#include <semaphore.h>
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t *sem);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

其中 `sem_t` 类型就代表一个信号灯,对信号灯的操作都通过它的指针来进行。

一、命名信号灯

`sem_open`, `sem_close`, `sem_unlink` 是专门用于命名信号灯的三个函数。下面分别对它们进行说明。

`sem_open` 函数用来新建或打开一个命名信号灯,其中参数 `name` 是一个字符串,表示信号灯的名字,它必须是 `/somename` 的形式,也就是说开头有一个斜杠,而中间没有斜杠。`oflag` 参数则用来指定打开信号灯的标志,有以下三个可能的取值。

- ◆ 0: 打开一个已有的命名信号灯。
- ◆ `O_CREAT`: 打开一个命名信号灯,如果不存在则创建它。
- ◆ `O_CREAT|O_EXCL`: 创建一个新的命名信号灯,如果已有同名信号灯存在则返回错误。

这些常数的定义与用 `open` 函数打开文件时是相同的,事实上,使用这些标志值时必须包含头文件 `fcntl.h`。

当 `oflag` 参数中包含 `O_CREAT` 时,必须同时通过参数 `mode` 提供新建信号灯的访问权限,以及通过 `value` 参数提供信号灯的初始值。

`sem_open` 函数返回一个指向信号灯的指针,随后的操作中可以用这个指针来代表这个信号灯。`sem_open` 函数在操作失败时返回 `SEM_FAILED`。

`sem_close` 函数用来关闭一个命名信号灯,当操作成功时它返回 0,否则返回 -1。与对文件的操作相似,进程不再使用一个信号灯后要将其关闭。



`sem_unlink` 函数则用来删除一个命名信号灯，当操作成功时它返回 0，否则返回 -1。删除一个信号灯后就可以用同样的名字再创建新的命名信号灯。

二、无名信号灯

`sem_init` 和 `sem_destroy` 是专门用于无名信号灯的两个函数。

使用无名信号灯时只需要直接生成一个 `sem_t` 型变量就可以了，然后用 `sem_init` 函数对它进行初始化。

初始化时，如果 `pshared` 参数为 0，则这个信号灯只能用于同一进程的各个线程之间的同步；如果 `pshared` 参数非 0，则这个信号灯可以用于不同进程间的同步，但信号灯还必须放在共享内存中，以使各个进程访问的是同一个信号灯。显然，无名信号灯只能用于有亲缘关系的进程间的同步。

`sem_init` 函数的 `value` 参数则用于设置信号灯的初始值。

`sem_destroy` 函数用于销毁一个无名信号灯，这样它所占用的资源就会被释放。

这两个函数操作成功时返回 0，失败时返回 -1。

三、信号灯操作

不管是命名信号灯还是无名信号灯，都可以用 `sem_post`、`sem_wait` 及 `sem_trywait` 函数进行操作。

`sem_post` 函数是信号灯的释放操作，`sem_wait` 函数是信号灯的获取操作。

`sem_trywait` 函数可视为获取操作的非阻塞版本，即当信号灯无法获取（值为 0）时，不是阻塞进程，而是以错误状态返回，这时 `errno` 变量的值被设为 `EAGAIN`。

这些函数操作成功时返回 0，失败时返回 -1。

下面使用命名信号灯的方式来解决上述 `mangle.c` 例程中的同步问题，新的代码如下：

```
/* 文件名: mangle_sem.c */
/* 说明: 使用信号灯解决进程同步问题 */

#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <semaphore.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
    int i, j;
    pid_t pid;
    const char *message = "Hello World!\n";
    int n = strlen(message) / 2;
    /* 打开命名信号灯，初始个数为 0 */
    sem_t *sem = sem_open("/mysem", O_CREAT, S_IRUSR | S_IWUSR, 0);
    assert(sem != NULL);
    pid = fork(); /* 生成新进程 */
    j = (pid == 0) ? 0 : n; /* 子进程从头开始输出，父进程从中间开始输出 */
```



```

    if (pid) sem_wait(sem); /* 父进程等待信号灯 */
    for (i = 0; i < n; i++) {
        write(STDOUT_FILENO, message + j + i, 1);
        sleep(1); /* 睡眠, 这样就会调度到其他进程 */
    }
    if (pid == 0) sem_post(sem); /* 子进程释放信号灯 */
    sem_close(sem); /* 关闭命名信号灯 */
    return 0;
}

```

在上述例程上, 信号灯的初始值被设为 0, 然后让父进程等待信号灯, 当子进程输出信息完毕后才释放信号灯, 这时父进程才可以继续执行并输出一句话的下半部分。

需要注意的是, 因为 POSIX IPC 接口并非 C 标准库的组成部分, 因此编译时必须与 `librt` 或 `libpthread` 共享库链接, 如:

```
gcc mangle_sem.c -Wall -o mangle_sem -lrt
```

或者:

```
gcc mangle_sem.c -Wall -o mangle_sem -lpthread
```

对于下面将要讲述的其他 POSIX IPC 接口也是这样的。

10.4.3 管道

这里的管道也可以称为无名管道, 以区别于后面将要讲述的有名管道。管道的创建与使用都很简单, 但只能用于有亲缘关系的进程间的通信。管道的编程接口如下:

```

#include <unistd.h>
int pipe(int pipefd[2]);

```

`pipe` 函数用于建立管道, 调用者必须传递一个有两个元素的整型数组的首地址作为参数。如果管道创建成功, 则函数返回后数组中存放的是管道的文件描述符。其中第一个描述符是以只读方式打开的, 称为管道的读端; 第二个描述符是以只写方式打开的, 称为管道的写端。向第二个描述符写入的数据将可以从第一个描述符中读出, 这就是管道的通信方式。

由于子进程继承父进程的文件描述符表, 因此它可以使用父进程中创建的管道与父进程通信。下面以一个例程来说明使用管道在父子进程之间进行通信的方法, 代码如下:

```

/* 文件名: pipe.c */
/* 说明: 管道使用例程 */

#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

#define MSGSIZE 8

```



```
char *msg = "hello";

int main(void)
{
    char inbuf[MSGSIZE];
    int p[2];
    pid_t pid;
    /* 打开管道 */
    if (pipe(p) == -1) {
        perror("pipe");
        exit(-1);
    }
    /* 生成子进程 */
    switch (pid = fork()) {
    case -1:
        perror("fork:");
        exit(2);
    case 0: /* 如果是子进程 */
        close(p[0]); /* 关闭管道读端 */
        /* 向写端写入 */
        write(p[1], msg, strlen(msg));
        write(p[1], "", 1); /* 写一个字符串结束符 */
        break;
    default: /* 父进程 */
        close(p[1]); /* 关闭管道写端 */
        /* 从读端读数据 */
        read(p[0], inbuf, MSGSIZE);
        printf("Parent: %s\n", inbuf);
        wait(NULL); /* 等待子进程结束 */
    }
    return 0;
}
```

在这个例程中，父子进程通过管道进行通信，子进程向管道的写端写入数据，父进程则从管道的读端将数据读出。注意管道是以一种数据流的方式进行传输的，并且数据是先进先出的。

10.4.4 命名管道

命名管道也称为 FIFO，它在形式上是一个特殊文件，可以独立于任何进程而存在。多个进程可以将 FIFO 文件打开进行读写，写入的数据会被按顺序读出来，这就是 FIFO 的通信方式。

FIFO 可以用 Shell 命令 `mkfifo` 建立，如：

```
mkfifo -m 666 myfifo
```

其中 `-m 666` 指定了文件的读写权限，`myfifo` 则是文件名。

在程序中则可以用 `mkfifo` 函数创建 FIFO，其接口头文件及原型如下：

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```



与创建文件一样,参数 `pathname` 是 FIFO 文件的路径,参数 `mode` 是其访问权限,创建成功返回 0,失败则返回 -1。

一个 FIFO 可以被打开、读、写和关闭,就像操作一个普通文件一样,使用相同的文件操作 API。FIFO 文件存在于文件系统中,但并不占据存储其文件内容的空间。

当进程打开一个 FIFO 时,内核为其分配一块空间进行数据的缓冲,这个空间的大小是固定的。当进程进行读操作时,如果没有相应的数据写入,那么它就会一直阻塞,直到有数据可读;反之,如果管道没有用于写入的空间,进程的写操作也会导致阻塞,直到数据被读走,留出足够的写入空间。

需要注意的是,如果以非阻塞方式打开 FIFO,则操作会阻塞直到管道的另一端也被打开。也就是说,以读方式打开 FIFO 时,如果 FIFO 已经由其他进程以写方式打开,则不会阻塞,如果 FIFO 还未以写方式打开过,则操作阻塞直到其他进程以写方式打开;反之,以写方式打开 FIFO 时,如果 FIFO 已经由其他进程以读方式打开,则不会阻塞,如果 FIFO 还未以读方式打开过,则操作阻塞直到其他进程以读方式打开。

以非阻塞方式打开 FIFO 也是可以的,这时,打开读的操作将总是成功,而打开写的操作仅在 FIFO 已经被打开读的情况下能成功。

下面对 FIFO 的读写特性分阻塞方式和非阻塞方式进行说明。

一、阻塞方式的读写操作

如果进程以读方式打开 FIFO 进行读操作,可能发生阻塞的情况有两种:一种情况是 FIFO 内有数据,但是其他进程正在读这些数据,另一种情况就是 FIFO 内没有数据。当 FIFO 内有了数据且没有其他进程在读时,阻塞将解除。另外要注意,因 FIFO 内没有数据而阻塞的情况只发生在第一次读操作时,后续的读操作在没有数据的情况下返回 0。

对于写操作,当要写入的数据量不大于 FIFO 的缓冲区时,系统将保证写入操作的原子性。也就是说,如果此时 FIFO 的空闲缓冲区不足以容纳要写入的字节数,则操作将阻塞,直到空闲缓冲区足够时才一次性写入。当要写入的数据量大于 FIFO 的缓冲区时,系统无法保证写入操作的原子性,只要 FIFO 的空闲缓冲区足够,就会试图写入数据,写完所有数据后,写操作返回。

二、非阻塞方式的读写操作

对于读操作来说,如果没有数据可读则返回 -1 并设置 `errno` 变量的值为 `EAGAIN`,提醒以后重试。

对于写操作来说,当要写入的数据量大于 FIFO 的缓冲区时,系统不保证写入操作的原子性,在写满所有 FIFO 空闲缓冲区后,写操作返回;当要写入的数据量不大于 FIFO 的缓冲区时,系统将保证写入的原子性。如果当前 FIFO 的空闲缓冲区能够容纳请求写入的字节数,写完后成功返回,如果不能则返回 -1 并设置 `errno` 变量的值为 `EAGAIN`,提醒以后重试。

下面以例程来说明 FIFO 的使用。这个例程包含三个独立的程序,它们的源文件分别是 `fifo.c`, `readfifo.c` 和 `writefifo.c`。

`fifo.c` 程序用于创建 FIFO,源码如下:

```
/* 文件名: fifo.c */
/* 说明: FIFO 例程 */
```



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
    /* 需要在命令行上输入 FIFO 文件名 */
    if (argc < 2) {
        printf("Usage: fifo fifo_name\n");
        exit(0);
    }
    /* 清除文件标志掩码 */
    umask(0);
    /* 创建 FIFO */
    if (mkfifo(argv[1], 0777))
    {
        perror("mkfifo");
        exit(1);
    }
    printf("success to create fifo %s\n", argv[1]);
    return 0;
}
```

readfifo.c 程序用于从 FIFO 读数据，源码如下：

```
/* 文件名: readfifo.c */
/* 说明: 读 FIFO */

#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

char work_area[32];

int main(void)
{
    int count = 0;
    char buf[128];
    int fd;
    /* 以读方式打开 FIFO 文件 */
    if ((fd = open("myfifo", O_RDONLY)) < 0) {
        perror("open fifo for read");
        exit(1);
    }
    printf("open fifo success to read\n");
    /* 循环读 */
    for (;;) {
        /* 需要用户输入一个字符 c 才开始读 */

```



```

while(strncmp("c", work_area, 1) != 0) {
    fgets(work_area, 32, stdin);
}
printf("ready to read fifo ..... \n");
/* 读一定数量的字节 */
count = read(fd, buf, 128);
printf("recv msg:%s\n", buf);
work_area[0]='\0';
}
close(fd);
return 0;
}

```

writefifo.c 程序用于向 FIFO 写入数据，源码如下：

```

/* 文件名: writefifo.c */
/* 说明: 写 FIFO */

#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
{
    int count = 0;
    int fd;
    int retry = 0;
    char buf[128] = "Hello world!";
    /* 以写方式打开 FIFO */
    if ((fd = open("myfifo", O_WRONLY)) < 0) {
        perror("open fifo for write");
        exit(1);
    }
    printf("open fifo success to write!\n");
    for (;;) {
        /* 写 FIFO */
        count = write(fd, buf, sizeof(buf));
        retry++;
        printf("Write %d bytes, %d times\n", count, retry);
    }
    close(fd);
    return 0;
}

```

从例程中可以看到，FIFO 由于有一个文件作为代表，克服了管道无名字的限制，所以可以用于没有亲缘关系的多个进程间通信。FIFO 的通信方式也是先进先出的数据流方式。

10.4.5 共享内存

一个进程不能简单地将自己的内存空间地址传递给其他进程使用，这是因为 Linux 操作系统

的内存保护机制或者说内存映射机制的限制。在一个进程内，指向一块内存的指针实际上是虚拟地址，而不是真正的物理内存地址，这个地址仅在当前进程内使用才是有效的。

但是如果通过某种方式能够实现多个进程访问同一块物理内存，那么进程之间的数据交换就可以通过读写内存来进行，这将是一种效率很高的通信方式，如图 10.9 所示。事实上，Linux 系统中确实有这样的机制来实现进程之间的内存共享。下面将介绍对 POSIX 共享内存的编程。

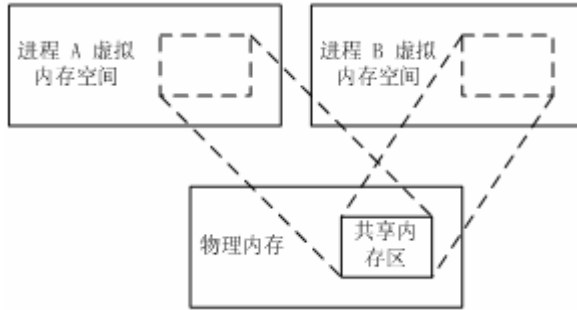


图 10.9 共享内存示意

POSIX 共享内存编程的接口头文件及函数原型如下：

```
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
void *mmap(void *addr, size_t length, int prot, int flags,
            int fd, off_t offset);
int munmap(void *addr, size_t length);
```

`shm_open` 函数用于打开一个已有的或建立一个新的共享内存区，其中 `name` 参数是一个字符串，用于指定共享内存区的名字，一般是 `/somename` 的形式，也就是说开头有一个斜杠，而中间没有斜杠。

`shm_open` 函数的 `oflag` 参数用于指定打开的方式，与打开文件类似，有如下取值。

- ◆ `O_RDONLY`：以只读方式打开共享内存区。
- ◆ `O_RDWR`：以可读可写方式打开共享内存区。
- ◆ `O_CREAT`：如果所要打开的共享内存区不存在则以这个名字建立一个新的共享内存区。
- ◆ `O_EXCL`：与 `O_CREAT` 一起使用，如果要建立的共享内存区已经存在则返回错误。
- ◆ `O_TRUNC`：将打开的共享内存区的长度截取为 0。

`shm_open` 函数的 `mode` 参数则用于指定共享内存区的访问权限。

如果 `shm_open` 函数执行成功，则返回值是一个文件描述符，也就是说，对共享内存区的操作也统一于 Linux 的文件操作之中；如果执行失败，则返回值是 -1。

使用 `shm_open` 函数创建的新共享内存区的长度是 0，一般需要马上用 `ftruncate` 函数调整它的大小，其接口头文件及函数原型如下：

```
#include <unistd.h>
#include <sys/types.h>
int ftruncate(int fd, off_t length);
```


这是一个文件操作，其中 `fd` 是要操作的文件描述符，`length` 指定调整以后的大小，如果操作成功则返回 0，否则返回 -1。

`shm_unlink` 函数用于删除一个已有的共享内存区。

实际上，一个共享内存区对应着特殊文件系统 `shm` 中的一个文件，一般来说，这个文件系统挂载在 `/dev/shm` 下。因此，创建一个共享内存区后，我们可以在 `/dev/shm` 目录下看到创建的文件。

使用共享内存区的关键在于理解 `mmap` 函数。本质上，它是一个将已打开的文件映射到当前进程内存空间的函数，但由于共享内存区也以文件描述符的形式出现，所以可以使用 `mmap` 函数。`mmap` 函数的各个参数的含义解释如下。

- ◆ `addr`: 指定映射得到的虚拟地址，一般来说，这只是一个建议值，因此可以设为 `NULL`，由内核自动选择合适的虚拟地址。
- ◆ `length`: 要映射的内存区域的长度，单位为字节。
- ◆ `prot`: 指定映射后内存的访问权限。
- ◆ `flags`: 指定内存映射的方式。
- ◆ `fd`: 要映射的文件描述符。
- ◆ `offset`: 文件中映射的起点位置。

这里 `prot` 参数有以下几个可能的取值。

- ◆ `PROT_READ`: 表示可读权限。
- ◆ `PROT_WRITE`: 表示可写权限。
- ◆ `PROT_EXEC`: 表示可执行权限。
- ◆ `PROT_NONE`: 表示不可访问。

这些值可以用“按位或”的方式组合起来使用。

`flags` 参数有以下两个基本取值。

- ◆ `MAP_SHARED`: 对文件的映射在进程间共享，也就是说，一个进程对共享内存的修改能够被其他进程看到。
- ◆ `MAP_PRIVATE`: 对文件的映射不在进程间共享，实际上是以“写时复制”的方式进行共享的，这时如果一个进程修改了共享内存，其他进程看不到所做的修改。

如果以上基本取值与 `MAP_ANONYMOUS` 用“按位或”的方式组合使用，则 `mmap` 函数可以进行所谓“匿名映射”，这时文件描述符可以是 -1，也就是说，不需要打开文件就可以进行映射。这种映射只能用于共享内存，可称之为匿名共享内存。显然，匿名共享内存只能用于有血缘关系的进程间的通信。

`mmap` 函数的返回值为映射得到的内存区域的首地址，进程可以直接对这块内存区域进行操作，如果映射失败则返回 `MAP_FAILED`。

`munmap` 函数与 `mmap` 函数的作用相反，用于取消内存映射，其中参数 `addr` 是要取消映射的内存区域的首地址，`length` 参数则是内存区域的长度。

对共享内存的基本使用过程如下。

- step 1** 某个进程使用 `shm_open` 函数创建一个共享内存区。
- step 2** 其他要进行通信的进程使用 `shm_open` 函数打开这个共享内存区。
- step 3** 各个进程分别将共享内存区映射到自己的虚拟地址空间。
- step 4** 各个进程分别对映射得到的内存区域进行操作。

如果是匿名共享内存，则使用过程如下。

- step 1** 父进程匿名映射一个共享内存区到自己的虚拟地址空间。
- step 2** 父进程创建子进程。
- step 3** 父子进程使用同样的内存首地址来操作共享内存区。

下面通过一个例程来说明共享内存的使用。在这个例程中，父子进程通过共享内存通信，子进程向共享内存写入数据然后退出，接着父进程从共享内存读出子进程写入的数据。例程代码如下：

```
/* 文件名: pmem.c */
/* 说明: POSIX 共享内存例程 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <sys/wait.h>

/* 输出错误信息并退出 */
void error_out(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
    int r;
    const char *memname = "/mymem";
    const size_t region_size = 1024;
    /* 打开共享内存文件 */
    int fd = shm_open(memname, O_CREAT | O_TRUNC | O_RDWR, 0666);
    if (fd == -1) error_out("shm_open");
    /* 将文件截取至指定的大小 */
    r = ftruncate(fd, region_size);
    if (r != 0) error_out("ftruncate");
    /* 将文件映射为内存 */
    void *ptr = mmap(0, region_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (ptr == MAP_FAILED) error_out("mmap");
    /* 关闭文件 */
    close(fd);
```

```

/* 产生新进程 */
pid_t pid = fork();
if (pid == 0) {
    /* 子进程向共享内存写入数据 */
    unsigned long *d = (unsigned long *) ptr;
    *d = 0xdeadbeef;
    exit(0);
} else {
    /* 父进程等待子进程结束并读共享内存 */
    int status;
    waitpid(pid, &status, 0);
    printf("The data child wrote is %lx\n", *(unsigned long *)ptr);
}
/* 取消内存映射 */
r = munmap(ptr, region_size);
if (r != 0) error_out("munmap");
/* 删除共享内存文件 */
r = shm_unlink(memname);
if (r != 0) error_out("shm_unlink");
return 0;
}

```

使用 POSIX 共享内存的程序必须与 `librt` 共享库链接，例如上述例程的编译命令如下：

```
gcc pmem.c -Wall -o pmem -lrt
```

由于是父子进程之间的通信，所以这个例程也可以使用匿名共享内存来实现。

在例程中，因为父进程使用 `waitpid` 函数等待子进程退出后才对共享内存进行操作，所以能够确保父子进程不会同时操作共享内存区，也就没有同步问题。如果存在多个进程并发对同一块共享内存区操作的情况，则需要仔细考虑同步问题，比如可以采用信号灯进行同步等手段。

10.4.6 消息队列

消息队列就是一个消息的链表，可以把一条消息看做一个记录，具有特定的格式以及特定的优先级，消息的发送和接收都以条为单位。

使用消息队列时，首先要创建消息队列，然后对消息队列有写权限的进程可以向队列中添加新消息，对消息队列有读权限的进程则可以从消息队列中读走消息。

一般来说，消息队列都有一个消息头，记录着消息队列的重要信息，这个结构是内核维护的，应用程序并不能直接修改使用。但是我们可以通过系统提供的设置消息队列属性的接口函数来更新或设置消息队列的属性。

下面将介绍 POSIX 消息队列的编程接口，所需头文件及主要的接口函数如下：

```

#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
mqd_t mq_close(mqd_t mqdes);
mqd_t mq_unlink(const char *name);
mqd_t mq_send(mqd_t mqdes, const char *msg_ptr,

```

```

    size_t msg_len, unsigned msg_prio);
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
    size_t msg_len, unsigned *msg_prio);
mqd_t mq_getattr(mqd_t mqdes, struct mq_attr *attr);
mqd_t mq_setattr(mqd_t mqdes, struct mq_attr *newattr,
    struct mq_attr *oldattr);

```

对消息队列的操作与文件操作有结构上的类似,但它们使用 `mqd_t` 型的消息队列描述符来标识一个消息队列,不一定能直接使用文件操作的接口函数,因此消息队列的各种操作都有专用的函数。

`mq_open` 函数可以打开或创建消息队列,其 `name` 参数是一个字符串,代表消息队列的名字,一般是 `/somename` 的形式,也就是说开头有一个斜杠,而中间没有斜杠。

`mq_open` 函数的 `oflag` 参数的取值如下。

- ◆ `O_RDONLY`: 以只读方式打开消息队列。
- ◆ `O_RDWR`: 以可读可写方式打开消息队列。
- ◆ `O_CREAT`: 如果所要打开的消息队列不存在则以这个名字建立一个新的消息队列。
- ◆ `O_EXCL`: 与 `O_CREAT` 一起使用,如果要建立的消息队列已经存在则返回错误。
- ◆ `O_NONBLOCK`: 将打开的消息队列设置为非阻塞操作方式。

如果 `oflag` 参数中包括 `O_CREAT` 标志,则必须提供 `mode` 参数,它代表新消息队列的访问权限,同时还要提供参数 `attr`,表示新消息队列的属性。`attr` 参数可以为 `NULL`,代表使用默认属性。

`mq_open` 函数操作成功时返回所打开的消息队列的描述符,失败时则返回 `-1`。

表示消息队列属性的类型 `struct mq_attr` 定义如下:

```

struct mq_attr {
    long mq_flags;        /* 标志,取值为 0 或 O_NONBLOCK */
    long mq_maxmsg;       /* 队列可容纳的最大消息数 */
    long mq_msgsize;      /* 每条消息的最大长度(字节数) */
    long mq_curmsgs;      /* 消息队列中当前的消息条数 */
};

```

`mq_close` 函数用于关闭由参数 `mqdes` 指定的消息队列,`mq_unlink` 函数用于删除由参数 `name` 指定的消息队列。它们在操作成功时返回 `0`,失败时返回 `-1`。

对消息队列的主要操作是发送消息和接收消息,分别由 `mq_send` 和 `mq_receive` 函数实现。发送消息使用 `mq_send` 函数,其各个参数及返回值的含义解释如下。

- ◆ `mqdes`: 用于发送的消息队列描述符。
- ◆ `msg_ptr`: 指向消息缓冲区。
- ◆ `msg_len`: 消息的长度,必须小于或等于消息队列的最大长度。
- ◆ `msg_prio`: 消息的优先级。
- ◆ 返回值: `0` 表示成功,`-1` 表示失败。

接收消息使用 `mq_receive` 函数,其各个参数及返回值的含义解释如下。

- ◆ mqdes: 用于接收的消息队列描述符。
- ◆ msg_ptr: 指向存放消息的缓冲区。
- ◆ msg_len: 缓冲区的长度。
- ◆ msg_prio: 用于返回接收到的消息的优先级。
- ◆ 返回值: 接收成功时是收到的消息的实际长度, 失败时是 -1。

如果队列以阻塞方式操作, 那么当进程从一个空的消息队列接收消息时就会被阻塞直到有消息可读。同样, 进程向一个已满的消息队列再发送消息也会被阻塞直到队列可以容纳新消息。

在消息队列打开后, 可以用 `mq_getattr` 函数来获得它的属性, 并用 `mq_setattr` 函数设置它的属性。`mq_getattr` 函数通过它的 `attr` 参数返回由参数 `mqdes` 指定的消息队列的属性, `mq_setattr` 函数则可以将参数 `mqdes` 指定的消息队列设置为参数 `newattr` 所指向的属性, 如果 `oldattr` 参数不是 `NULL`, 还会把原来的属性放在 `oldattr` 指向的变量中。这两个函数在操作成功时返回 0, 失败时返回 -1。

需要指出的是, `struct mq_attr` 的 `mq_maxmsg` 和 `mq_msgsize` 成员仅在 `mq_open` 函数中使用才是有效的。也就是说, 一个消息队列的最大消息数和最大消息长度是在它创建时决定的, 不能随意修改。使用 `mq_setattr` 函数设置消息队列属性时起作用的仅仅是 `mq_flags` 成员, 即只能改变队列的阻塞或非阻塞操作方式。`mq_curmsgs` 成员则仅在返回消息队列属性时有意义。

POSIX 消息队列实际上对应着 `mqueue` 文件系统中的文件, 这个文件系统默认并不会自动挂载在系统中, 可以通过类似以下的命令挂载:

```
sudo mkdir /dev/mqueue
sudo mount -t mqueue none /dev/mqueue
```

如上挂载之后就可以在 `/dev/mqueue` 目录中看到消息队列所对应的文件。

下面通过例程来说明 POSIX 消息队列的使用, 代码如下:

```
/* 文件名: pmsgq.c */
/* 说明: POSIX 消息队列例程 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <sys/wait.h>

#define MSGSIZE 128

/* 包装消息 */
struct message {
    char mtext[MSGSIZE];
};

/* 发送消息 */
static int send_msg(mqd_t qid, int pri, const char text[])
```



```

{
    int r = mq_send(qid, text, strlen(text)+1, pri);
    if (r == -1) {
        perror("mq_send");
    }
    return r;
}

/* 消息发送者 */
static void producer(mqd_t qid)
{
    /* 发送低优先级消息 */
    send_msg(qid, 1, "This is my first message.");
    send_msg(qid, 1, "This is my second message.");
    /* 发送高优先级消息 */
    send_msg(qid, 3, "No more messages.");
}

/* 消息接收者 */
static void consumer(mqd_t qid)
{
    struct mq_attr mattr;
    do {
        u_int pri;
        struct message msg;
        ssize_t len;
        len = mq_receive(qid, (char *)&msg, sizeof(struct message), &pri);
        if (len == -1) {
            perror("mq_receive");
            break;
        }
        printf("got pri %d '%s' len=%d\n", pri, msg.mtext, len);
        /* 检查是否有更多的消息 */
        int r = mq_getattr(qid, &mattr);
        if (r == -1) {
            perror("mq_getattr");
            break;
        }
    } while (mattr.mq_curmsgs); /* 如果没有消息则退出循环 */
}

int main(void)
{
    mqd_t qid;
    struct mq_attr attr;
    char *qname = "/msgq"; /* 队列的名字 */
    memset(&attr, 0, sizeof(attr));
    attr.mq_maxmsg = 5;
    attr.mq_msgsize = MSGSIZE;
    qid = mq_open(qname, O_CREAT|O_RDWR, 0666, &attr); /* 创建队列 */
    if (qid < 0) {

```



```
perror("mq_open");
return -1;
}
if (!fork()) {
    producer(qid);
    exit(0);
}
wait(NULL); /* 等待子进程结束 */
consumer(qid);
mq_unlink(qname); /* 删除队列 */
return 0;
}
```

注意编译消息队列相关的程序时也必须与 **librt** 共享库链接。

在主函数中，首先创建了一个消息队列，然后创建了一个子进程，在子进程中发送了若干条消息后退出，随后父进程从消息队列中接收消息并显示在屏幕上。

程序的运行结果如下：

```
got pri 3 'No more messages.' len=18
got pri 1 'This is my first message.' len=26
got pri 1 'This is my second message.' len=27
```

从结果中可以看到消息优先级的作用，后发送的消息如果优先级较高则有可能先被收到，同一优先级的消息传送则满足先进先出的原则。这个例程也说明消息队列能够容纳消息，甚至在发送者进程退出后也保留它所发送的消息。



第 11 章 socket 编程

TCP/IP 是网络中应用最广泛的协议，因此几乎所有现代的操作系统都支持 TCP/IP 协议，包括 Linux。在嵌入式系统上选择 Linux 作为操作系统的一大理由就是它对网络协议具有的良好支持。

Linux 对 TCP/IP 协议的实现放在内核中，应用程序通过 socket 编程接口来使用这些协议。当然，socket 编程接口并非仅仅局限于对 TCP/IP 协议的支持。socket 在很多地方又被翻译为套接字。

在本章中，首先介绍的是计算机系统上的网络协议层次结构，然后介绍 socket 网络编程接口，主要是其在 TCP(Transmission Control Protocol, 传输控制协议)和 UDP(User Datagram Protocol, 用户数据报协议)方面的使用，最后通过例程来说明如何使用 socket 接口进行网络编程。

11.1 网络协议层次模型

如图 11.1 所示是 OSI 定义的网络协议模型及其与实际常用的网络协议的对照。下面对其中几个基本层次进行简要的说明。

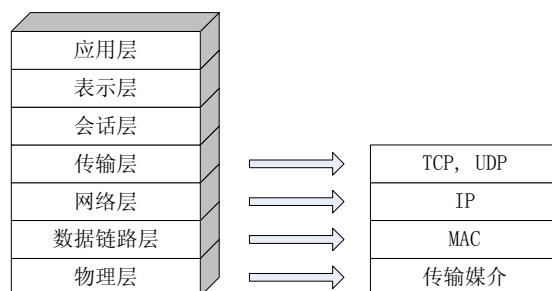


图 11.1 网络层次模型

一、物理层

物理层主要负责物理信号的传送，它是所有网络通信的基础。物理层不仅仅是指进行传输所必需的硬件设施，而且还包括使信号得以发送和接收的协议，如信号的调制方式等。一般来说，物理层的传输是不可靠的，所发送的信号有可能被扭曲或丢失，还有可能在接收端产生误码。一般来讲，这一层的功能都是由硬件设备实现的。

二、数据链路层

数据链路层简称链路层，主要负责数据的无差错传输。通过在传输的数据中加入校验字节，接收端可以进行检错或纠错，以在一定程度上保证数据的正确性。数据链路层并不保证传输的可靠性，所发送的数据有可能丢失。这一层的功能可由硬件设备（或设备上的软件）实现，也可由软件实现。我们熟知的以太网协议实际上既包含物理层的内容，也包含数据链路层的内容，由于

这一协议使用 MAC (Media Access Control, 媒介访问控制) 地址对所访问的节点进行控制, 也常称为 MAC 协议。

三、网络层

利用数据链路层可以实现网络中直接相连的两个节点间互传数据, 而网络层在它的基础上引进了路由与转发的功能, 因而可以在逻辑上不直接相连的节点间互传数据。网络层的更大意义在于, 它可以使不同底层 (物理层和数据链路层) 的网络之间互联互通。IP 协议就属于这一层次。网络层也不能保证数据传送的可靠性, 所传送的数据有可能丢失, 各个数据包的接收顺序也无法保持与发送端一致。

四、传输层

传输层在网络层的基础上加入了更多的控制。有了传输层以后, 网络中的节点可以同时与其他节点建立多个数据通道, 它们之间互不干扰。常见的传输层协议是 TCP 和 UDP。

在 Linux 操作系统上, 传输层及以下的层次都实现在内核中, 上层所采用的协议则由应用程序根据需要决定, 因此实际上传输层以上的层次并不一定被实际使用。

在各个层次间, 上层的协议必须依赖于下层的功能才能实现通信, 各个层次有堆叠关系, 因此常被称为协议栈。TCP/IP 协议栈就是指基于 IP 协议进行传输的 TCP 协议。

11.2 socket 编程接口

socket 是一种网络编程的标准接口。在 Linux 系统中, 从编程的角度看它实际上就是文件描述符, 因此适用于文件的各种操作, 如读、写、关闭、重定向等也都适用于 socket, 但这个文件毕竟不是普通文件, 因此也有自己特有的一些操作接口。

与 socket 编程有关的主要头文件有以下两个:

```
#include <sys/types.h> /* 并非必须 */
#include <sys/socket.h>
```

有些编程接口需要包含额外的头文件, 将在介绍接口时一并指出。

11.2.1 打开 socket

打开 socket 实际上就是获得一个文件描述符, 其接口函数原型如下:

```
int socket(int domain, int type, int protocol);
```

其各个参数及返回值的含义解释如下。

- ◆ domain: 表示进行通信的协议族。
- ◆ type: 表示 socket 的类型。
- ◆ protocol: 表示 socket 所用的协议。
- ◆ 返回值: socket 的描述符, -1 则表示失败。

常见的一些协议族列举如下。

- ◆ PF_UNIX, PF_LOCAL：用于本地进程间通信。
- ◆ PF_INET：IPv4 协议。
- ◆ PF_INET6：IPv6 协议。
- ◆ PF_NETLINK：用于内核与应用程序通信。
- ◆ PF_PACKET：用于访问底层数据。

常用的 socket 类型有以下几种。

- ◆ SOCK_STREAM：数据流类型。这种类型的 socket 以流的方式传输数据，类似于管道，数据以字节为单位顺序传输，没有数据包的界限。
- ◆ SOCK_DGRAM：数据报类型。这种类型的 socket 以数据报的方式传输数据，数据被组织成不同长度的数据包，以包为单位进行传输。
- ◆ SOCK_RAW：原始类型。这种类型的 socket 可以访问底层的原始数据。

不同的协议族可以支持不同的 socket 类型。即使是相同类型的 socket，如果建立在不同的协议族上，具体的功能也有所不同，如表 11.1 所示是常见协议族与 socket 类型的组合及它们实际的作用。

表 11.1 协议族与 socket 类型

协议族	socket 类型	作用
PF_UNIX	SOCK_STREAM	本地进程间数据流通信
PF_LOCAL	SOCK_DGRAM	本地进程间数据报通信
PF_INET	SOCK_STREAM	基于 IPv4 的数据流通信，使用 TCP 协议
	SOCK_DGRAM	基于 IPv4 的数据报通信，使用 UDP 协议
	SOCK_RAW	直接访问 IPv4 网络层数据
PF_INET6	SOCK_STREAM	基于 IPv6 的数据流通信，使用 TCP 协议
	SOCK_DGRAM	基于 IPv6 的数据报通信，使用 UDP 协议
	SOCK_RAW	直接访问 IPv6 网络层数据
PF_NETLINK	SOCK_DGRAM	内核与应用程序间的通信
	SOCK_RAW	
PF_PACKET	SOCK_DGRAM	直接访问链路层原始数据，不包含链路层信息头
	SOCK_RAW	直接访问链路层原始数据，包含链路层信息头

从实用的角度出发，本章中将主要介绍基于 PF_INET 协议族的 SOCK_STREAM 和 SOCK_DGRAM 两种类型 socket 的使用。如不再说明，所有叙述都是针对 PF_INET 协议族进行的。

socket 接口函数原型中 protocol 参数用于表示所用的协议，常用值有以下两个。

- ◆ IPPROTO_TCP：表示 TCP 协议。
- ◆ IPPROTO_UDP：表示 UDP 协议。

事实上，当协议族确定为 PF_INET 后，SOCK_STREAM 类型的 socket 一定使用 TCP 协议，

而 `SOCK_DGRAM` 类型的 `socket` 一定使用 `UDP` 协议,在这种情况下,`protocol` 参数可以传入 `0`。也就是说,我们一般用下面的语句打开数据流类型的 `socket`:

```
tcp_socket = socket(PF_INET, SOCK_STREAM, 0);
```

而用下面的语句打开数据报类型的 `socket`:

```
udp_socket = socket(PF_INET, SOCK_DGRAM, 0);
```

`TCP` 协议与 `UDP` 协议的区别不仅仅在通信方式上,由于实现上的差异,它们的特性有很大的差异,表 11.2 中对比了这两种传输层协议的一些特性。

表 11.2 TCP 与 UDP 特性对比

TCP	UDP
有连接	无连接
数据流方式	数据报方式
服务器/客户端模式	点到点模式
可靠传输	不可靠传输
保证收到的数据正确	保证收到的数据包正确
保证发送成功的数据都能被收到	发送成功的数据包也不一定能被收到
保证数据接收的顺序与发送时一致	接收数据包的顺序不一定与发送时一致
数据不可能重复收到	数据包有可能重复收到
实现复杂	实现简单

由于通信手段的多样性及应用场景的复杂性,对通信服务的需求是不一样的,因此应根据实际情况来选择通信的类型。`TCP` 协议由于其可靠性,在各种应用中使用得十分广泛,但并不是说 `UDP` 协议就无用武之地,例如在 `bootloader` 中用于下载文件的 `TFTP` 协议就是基于 `UDP` 的。

11.2.2 socket 地址

网络通信程序中,通信双方需要有确定的通信地址。`Linux` 系统上定义了一个通用的地址结构 `struct sockaddr`,这里将其称为 `socket` 地址。许多 `socket API` 需要这种类型的数据作为参数,其定义如下:

```
struct sockaddr {
    sa_family_t sa_family;
    char sa_data[14];
};
```

其中 `sa_family` 成员是地址的类型,不同的协议族有不同的地址类型,对应于 `PF_INET` 协议族的地址类型为 `AF_INET`。`sa_data` 为 14 个字节的字符数组,这样,不管地址是何种类型,只要不超过 14 个字节就可以存放在这个字符数组内。

实际上,对于 `IPv4` 协议,我们经常用另外一个数据类型表示地址,其接口头文件及定义如下:

```
#include <netinet/in.h>
```



```

struct sockaddr_in {
    sa_family_t    sin_family;    /* 地址类型必须是 AF_INET */
    uint16_t       sin_port;      /* 端口号, 必须是网络字节序 */
    struct in_addr sin_addr;      /* IP 地址 */
};

struct in_addr {
    uint32_t       s_addr;        /* IP 地址, 必须是网络字节序 */
};

```

可以看出, 整个地址由一个 16 位的端口号和一个 32 位的 IP 地址组成。这个数据类型可以被强制转换为 (struct sockaddr) 型使用。

在用户界面上表示 IP 地址时, 一般都使用以小数点分隔的字符串形式表示, 每个被分隔的部分是一个取值不超过 255 的整数, 如 192.168.1.10, 这在编程时是以字符串表示的。而在进行 socket 编程时, 表示 IP 地址的数据要求是一个 32 位整数, 这就存在一个相互转换的问题。系统已经提供了进行转换的 API, 其接口头文件与函数原型如下:

```

#include <netinet/in.h>
#include <arpa/inet.h>
int inet_aton(const char *cp, struct in_addr *inp);
char *inet_ntoa(struct in_addr in);

```

inet_aton 函数可以将字符串参数 cp 所表示的 IP 地址转换到 inp 指向的变量中, 转换成功则返回非 0 值, 否则返回 0。

inet_ntoa 函数可以将参数 in 所表示的 IP 地址转换为字符串, 并返回其指针。注意转换后的字符串放在函数内部的一块静态缓冲区中, 会被后续的转换操作覆盖。

这两个函数所处理的 32 位整型 IP 地址都是网络字节序, 不需要再做字节序转换。

11.2.3 网络字节序

我们知道, 数据在内存中的存储有所谓大端和小端的概念, 这个问题在网络通信时变得突出起来, 因为此时通信的双方可能采用的是不同的存储方式。举个例子, 由于网络传输以字节为单位, 如果要传输一个整数, 发送方必须将内存中的 4 个字节按顺序发出。接收方接收到这 4 个字节的数据后, 将按照自己对数据存储的方式理解。如果收发双方是相同存储方式的计算机, 则对这个整数的理解是一致的。但如果双方所采用的字节序不同, 则相同的 4 个字节就会被理解为不同的整数。

因此, 在异构环境下的网络通信必须解决由于数据存储格式不同带来的整型及短整型数据的表达问题, 最简单的方法就是约定网络上字节传送采用相同的方式, 实际上采用了大端的传输方式, 这就是所谓网络字节序。为此, Linux 系统提供了几个辅助函数进行整型数的字节序转换, 其接口头文件及函数原型如下:

```

#include <netinet/in.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);

```



它们的参数是被转换的数据，返回值是转换后的数据，各自的作用解释如下。

- ◆ htonl: 用于将 32 位整数从本地字节序转换为网络字节序。
- ◆ htons: 用于将 16 位整数从本地字节序转换为网络字节序。
- ◆ ntohl: 用于将 32 位整数从网络字节序转换为本地字节序。
- ◆ ntohs: 用于将 16 位整数从网络字节序转换为本地字节序。

这些函数的实现是与平台相关的。事实上，在大端的主机上，这些函数直接返回数据本身；而在小端的主机上，这些函数会将字节序颠倒后返回。必须注意，它们转换字节序时并不修改参数本身。编程时，只有那些要在网络上传输的、并且为整型或短整型的信息才有网络字节序的问题。例如在数据结构 (struct sockaddr_in) 中，表示 IP 地址和端口的数据就要求是网络字节序。

11.2.4 socket 与地址的绑定

一个 socket 地址可以唯一地表示一个通信的端点。在接收数据前，socket 必须与某个地址绑定，表示要接收目标为这个地址的数据。绑定操作的函数原型如下：

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

其各个参数的含义解释如下。

- ◆ sockfd: 被绑定的 socket 描述符。
- ◆ addr: 所绑定的 socket 地址。
- ◆ addrlen: 地址数据的长度。

通常用于绑定的代码如下：

```
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr)); /* 将数据清 0 */
addr.sin_family = AF_INET; /* 地址类型为 IPv4 */
addr.sin_addr.s_addr = htonl(INADDR_ANY); /* IP 地址为任意 */
addr.sin_port = htons(port); /* port 是指定的端口 */
result = bind(sockfd, (struct sockaddr *)&addr, sizeof(addr));
```

这里 INADDR_ANY 实际上就是 0，它表示绑定在任意 IP 地址上，也可以指定主机的任意一个有效 IP。如果 socket 没有地址可重用的属性，则它所绑定的 socket 地址不能再被其他 socket 绑定，也就是说一个 socket 地址只能被一个 socket 绑定。



SOCK_STREAM 和 SOCK_DGRAM 类型的 socket 各自有自己的端口空间，两个不同类型的 socket 绑定在同一端口号上并不冲突。

11.2.5 侦听

在 TCP 协议服务器端编程时，有一个特殊的步骤称为侦听。侦听操作将使一个 socket 进入被动状态，用来接收来自其他主机的连接请求。侦听操作的函数原型如下：

```
int listen(int sockfd, int backlog);
```

其各个参数及返回值的含义解释如下。

- ◆ **sockfd**: 侦听的 **socket** 描述符。
- ◆ **backlog**: 连接请求队列的最大长度, 如果一个新的连接请求到达并且队列已满, 则请求将被拒绝。
- ◆ **返回值**: 0 表示成功, -1 表示有错误发生。



进行侦听操作的 **socket** 必须是 **SOCK_STREAM** 类型。

在侦听之前, **socket** 必须与某个 **socket** 地址绑定。需要注意的是, 不能有两个 **socket** 同时在一个地址上侦听。如果进行侦听时, 所绑定的地址上已有另外一个 **socket** 在侦听, 则操作会失败。

11.2.6 接受连接请求

已经启动侦听的 **socket** 可以接受连接请求, 所用的接口函数原型如下:

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

其各个参数及返回值的含义解释如下。

- ◆ **sockfd**: 要接受连接请求的 **socket** 描述符。
- ◆ **addr**: 用于返回对端的 **socket** 地址。
- ◆ **addrlen**: 保存 **socket** 地址的数据类型的大小, 它既是输入型参数又是输出型参数, 调用函数前先将 ***addrlen** 变量赋为 **addr** 参数指向的变量的大小, 函数返回后它的值是地址数据的实际大小。
- ◆ **返回值**: 成功则返回与对端连接的新 **socket** 的描述符, -1 表示出错。

accept 操作类似于读数据的操作, 有阻塞方式和非阻塞方式的区别。如果 **sockfd** 以阻塞方式操作, 那么当它的连接请求队列为空的时候, 对 **accept** 函数的调用会阻塞当前进程的执行; 如果 **sockfd** 以非阻塞方式操作, 那么当它的连接请求队列为空的时候, 对 **accept** 函数的调用会失败, 并且 **errno** 变量的值被设为 **EAGAIN**。

理解 **accept** 操作的关键在于它会返回一个新的 **socket** 描述符, 这是与对端连接成功的一个 **socket**, 可以通过它与对端进行通信, 通信的方式是全双工的, 既可以发送也可以接收。原来进行侦听的 **socket** 仍然存在并且状态不变, 只是队列中少了一个连接请求。

11.2.7 连接

对应于服务器端的 **accept** 操作, 客户端需要有发起连接请求的操作, 其接口函数原型如下:

```
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

其各个参数及返回值的含义解释如下。

- ◆ sockfd: 发起连接的 socket 描述符。
- ◆ serv_addr: 连接的对端 socket 地址 (服务器地址和端口)。
- ◆ addrlen: 地址数据的长度。
- ◆ 返回值: 连接成功返回 0, 连接失败或出错则返回 -1。

连接成功之后, sockfd 这个 socket 就可以用来与服务器端通信了, 通信的方式是全双工的。连接操作是以阻塞方式执行的, 当一个进程调用 connect 函数向服务器发起请求时, 执行流程会阻塞直到得到服务器的响应或超时。

11.2.8 关闭和切断连接

socket 描述符既然也是文件描述符, 关闭它与关闭文件就使用相同的操作 close, 其接口头文件与函数原型如下:

```
#include <unistd.h>
int close(int fd);
```

对于有连接的 socket, 还可以用下面的函数切断连接:

```
int shutdown(int sockfd, int how);
```

其各个参数及返回值的含义解释如下。

- ◆ sockfd: 要被切断连接的 socket 描述符。
- ◆ how: 切断的方式。
- ◆ 返回值: 0 表示操作成功, -1 表示操作失败。

how 参数的取值及作用如下。

- ◆ SHUT_RD: 切断接收方向。
- ◆ SHUT_WR: 切断发送方向。
- ◆ SHUT_RDWR: 切断接收与发送方向。

被切断连接的 socket 不能再进行通信, 但仍然处于打开状态, 可以进行其他操作。



socket 被关闭后, 如果已经没有任何进程在使用, 则同时会切断连接, 因此编程时一般不需要显式地调用 shutdown 函数。

11.2.9 发送数据

对于 socket 描述符完全可以使用文件操作的 write 函数来发送数据, 也可以用一些专门的函数, 这些函数有更多的参数, 因此使用起来更加灵活。

发送数据的函数原型如下:

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

其各个参数及返回值的含义解释如下。

- ◆ sockfd: 要发送数据的 **socket** 描述符。
- ◆ buf: 指向存放待发送数据的缓冲区。
- ◆ len: 要发送的字节数 (缓冲区大小)。
- ◆ flags: 一些标志位, 可以改变函数的操作行为。
- ◆ 返回值: 发送成功的字节数, -1 表示有错误发生。

这个函数通常用于有连接的 **socket**。与 **write** 函数相比, 它多了一个 **flags** 参数, 这个参数的一些常用取值及作用如下。

- ◆ **MSG_DONTWAIT**: 进行非阻塞的操作。
- ◆ **MSG_NOSIGNAL**: 当连接中断时, 不要发送 **SIGPIPE** 信号 (操作仍会返回错误, 并且 **errno** 变量的值被设为 **EPIPE**)。

这些值可以用按位或的方式组合起来使用。

当 **socket** 是无连接类型 (**SOCK_DGRAM**) 时, 发送数据时必须指明目标地址, 这可以用另外一个发送数据的函数, 原型如下:

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,  
               const struct sockaddr *to, socklen_t tolen);
```

与 **send** 函数相比, 它多了两个参数 **to** 和 **tolen**, **to** 指向目标 **socket** 地址, **tolen** 则是地址数据的长度。其他参数和返回值的含义与 **send** 函数是一致的。

即使是无连接类型的 **socket**, 也可以先使用 **connect** 函数设置一个默认的目标地址, 然后就可以用 **send** 函数进行发送了。必须注意, 这里用 **connect** 函数并不会发起连接请求, 而仅仅是设置一个默认地址而已。

与 **write** 函数一样, 发送数据的操作也有阻塞和非阻塞方式的差别。

11.2.10 接收数据

接收数据可以使用文件操作的 **read** 函数, 也可以用专门的函数, 其中一个原型如下:

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

其各个参数及返回值的含义解释如下。

- ◆ sockfd: 要接收数据的 **socket** 描述符。
- ◆ buf: 指向存放接收到数据的缓冲区。
- ◆ len: 要接收的字节数 (缓冲区大小)。
- ◆ flags: 一些标志位, 可以改变函数的操作行为。
- ◆ 返回值: 接收成功的字节数, -1 表示有错误发生。

与 `write` 函数相比，这个函数多了一个 `flags` 参数，其常用取值及作用如下。

- ◆ `MSG_DONTWAIT`：进行非阻塞的操作。
- ◆ `MSG_NOSIGNAL`：当连接中断时，不要发送 `SIGPIPE` 信号（操作仍会返回错误，并且 `errno` 变量的值被设为 `EPIPE`）。
- ◆ `MSG_PEEK`：让接收到的数据仍然保留在接收缓冲区内，下次操作这些数据仍会被收到。
- ◆ `MSG_TRUNC`：仅用于无连接类型的 `socket`，让返回值是包的实际大小，而不是接收到的数据的字节数（两者是不同的，因为如果包的长度大于所提供的缓冲区的长度，多余的字节将被截去）。

这个函数通常用于有连接的 `socket`，因为与它通信的对端的地址是已知的。另外一个函数可以在接收数据的同时得到发送者的地址，因此常用于无连接类型的 `socket`，其原型如下：

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

与 `recv` 函数相比，它多了两个参数 `from` 和 `fromlen`，`from` 指向一个变量，用于返回发送者的 `socket` 地址，`fromlen` 指向的变量则表示地址数据的长度，在调用函数前应赋值为 `from` 所指变量的大小，调用返回后则是地址数据的实际大小。

与 `read` 函数一样，接收数据的操作也有阻塞和非阻塞方式的差别。

11.2.11 使用 socket 选项

`socket` 有很多选项可以控制它的行为，这些选项需要用两个专门的函数来操作。

获取 `socket` 选项值的函数原型如下：

```
int getsockopt(int sockfd, int level, int optname,
               void *optval, socklen_t *optlen);
```

其各个参数及返回值的含义解释如下。

- ◆ `sockfd`：要操作的 `socket` 描述符。
- ◆ `level`：选项所在的网络层次。
- ◆ `optname`：由整数代表的选项名称。
- ◆ `optval`：指向存放选项值的内存。
- ◆ `optlen`：指向表示选项值长度的变量，调用前应将其赋值为所允许的最大长度（`optval` 指向的缓冲区的长度），函数返回后为选项值的实际长度。
- ◆ 返回值：0 表示成功，-1 表示失败。

设置 `socket` 选项值的函数原型如下：

```
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
```

与 `getsockopt` 函数相比，首先它的 `optval` 参数变为只读指针型，因为函数不会对缓冲区中

的数据进行修改，它指向的数据是要设置的选项值；其次 `optlen` 参数不再是指针型，它本身就表示选项值的长度；其余参数和返回值的含义都相同。

参数 `level` 表示选项所在的网络层次，对于 `socket` 本身的选项，`level` 参数的取值应是 `SOL_SOCKET`，否则应是网络协议的编号，例如，`IPPROTO_TCP` 表示要操作 `TCP` 协议层的选项，而 `IPPROTO_IP` 表示要操作 `IP` 协议层的选项。不同的网络层次有不同的选项。

选项的名称实际上是一个整数，这些整数都有一个方便记忆的宏定义。选项的值的类型则根据选项名称的不同而不同，因此用一个 `(void *)` 型指针代表。

下面将介绍一些常用的选项，它们都是 `socket` 层次的选项。

11.2.11.1 设置地址可重用

`socket` 设置了地址可重用以后，它所绑定的 `socket` 地址就可以同时被其他 `socket` 绑定。但是要注意，同时绑定在一个地址上的 `socket` 仍然只有一个能进行侦听。

设置地址可重用的基本方法如下：

```
int val = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, (void *)&val, sizeof(val));
```

这是一个开关型的选项，将选项值设为 `0` 即可关闭地址可重用。

出于安全考虑，被 `socket` 绑定的地址在这个 `socket` 被关闭后一段时间内仍然保持被绑定的状态，也就是不能为其他 `socket` 所绑定。这将对服务器程序的重新启动造成影响。设置地址可重用后，服务器程序就可以立刻重启，不必等待上次退出前绑定的地址重新成为可用状态。

11.2.11.2 设置缓冲区大小

每个 `socket` 在系统的底层都有发送和接收缓冲区，调整这些缓冲区的大小将影响通信的性能。设置发送缓冲区大小的基本方法如下：

```
int val = 1024;
setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF, (void *)&val, sizeof(val));
```

而设置接收缓冲区大小的基本方法如下：

```
val = 128;
setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, (void *)&val, sizeof(val));
```

这里要注意的是，设置的缓冲区大小会被内核加倍，并且用 `getsockopt` 函数得到的大小是加倍后的值。在上述例子中，发送缓冲区实际被设置为 `2048` 字节，接收缓冲区实际被设为 `256` 字节。

11.2.11.3 设置超时

设置超时的意义在于，当以阻塞方式发送或接收数据时，进程的等待时间有一个上限。如果超过这个时间，则不管是否读到或写入数据，函数都会返回，这样就避免了进程一直阻塞在某个操作上。超时选项有两个，分别对应发送与接收操作，设置的基本方法如下：

```
struct timeval tv;
tv.tv_sec = timeout; /* 超时的秒数 */
```

```
tv.tv_usec = 0; /* 超时的微秒数 */
setsockopt(sockfd, SOL_SOCKET, SO_SNDTIMEO, &tv, sizeof(tv));
setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
```

其中表示时间长度的数据类型定义如下：

```
struct timeval {
    long    tv_sec;        /* 秒 */
    long    tv_usec;       /* 微秒 */
};
```

对于发送和接收操作，如果已经收到部分数据，则超时后仍然返回收到的字节数；如果超时后还没有收到数据，则返回 -1，并且变量 `errno` 的值被设为 `EAGAIN`，类似于非阻塞操作那样。

如果超时值设为 0，则相当于取消超时。



`connect` 操作受发送超时的影响。

11.2.12 阻塞与非阻塞操作

默认情况下 `socket` 是以阻塞方式进行操作的。如果要改为非阻塞方式，则有两种方法可以采用。

一种方法是利用发送和接收操作本身的参数，如：

```
send(sockfd, buf, len, MSG_DONTWAIT);
```

其中 `MSG_DONTWAIT` 标志将使本次操作成为非阻塞的。

另一种方法是利用 `socket` 本身也是文件的特性，修改文件的标志位。与打开文件的 `open` 函数不同，打开 `socket` 的 `socket` 函数本身没有设置标志位的参数，因此必须在打开之后对标志位进行修改，这可以使用 `fcntl` 函数实现。这是一个专门用来对文件描述符进行各种查询和设置操作的函数，其接口头文件及原型如下：

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
int fcntl(int fd, int cmd, struct flock *lock);
```

其中参数 `fd` 是要操作的文件描述符，参数 `cmd` 代表所要进行的操作，参数 `arg` 的含义与所要进行的操作有关，参数 `lock` 是进行文件锁相关的操作时所需的一个参数，函数的返回值与所要进行的操作有关。这里不再详叙 `fcntl` 函数的用法，而是直接给出如下的修改文件标志位的方法：

```
int flag = fcntl(sockfd, F_GETFL); /* 得到旧的标志位 */
flag |= O_NONBLOCK; /* 增加非阻塞标志位 */
fcntl(sockfd, F_SETFL, flag); /* 设置新的标志位 */
```



当文件设置了非阻塞标志位时，后续的所有相关操作都默认为非阻塞的。

11.2.13 可靠的发送与接收操作

使用阻塞方式进行读写操作时，要注意读写过程被信号打断的问题，特别是当读写一个数据流式 `socket` 时，如果连接已中断，则会发送 `SIGPIPE` 信号，此信号的默认处理是使进程退出，通常这不是期望的结果。解决这个问题的办法之一是捕获 `SIGPIPE` 信号，另一种方法是在发送或接收时传入 `MSG_NOSIGNAL` 参数，这样就不会发送 `SIGPIPE` 信号了。举例如下：

```
send(sockfd, buf, len, MSG_NOSIGNAL);
```

但即使这样，其他信号仍然会被发送，并且导致系统调用提前返回。如果要可靠地发送指定数量的字节，必须循环进行发送，举例如下：

```
int send_byte(int sock, const char *buf, int len)
{
    int rc;
    int byte;
    for (byte = 0; byte < len; byte += rc) {
        rc = send(sock, buf+byte, len-byte, MSG_NOSIGNAL);
        if (rc < 0 && errno != EINTR) {
            byte = -1;
            break;
        }
    }
    return byte;
}
```

这个函数可以使用 `socket` 描述符 `sock` 可靠地发送 `len` 个字节，`buf` 参数指向存放待发送数据的缓冲区。而可靠地接收指定数量字节的函数可实现如下：

```
int recv_byte(int sock, char *buf, int len)
{
    int rc;
    int byte;
    for (byte = 0; byte < len; byte += rc) {
        rc = recv(sock, buf+byte, len-byte, MSG_NOSIGNAL);
        if (rc == 0) break;
        if (rc < 0 && errno != EINTR) {
            byte = -1;
            break;
        }
    }
    return byte;
}
```

其中 `sock` 参数是用于接收的 `socket` 描述符，参数 `buf` 指向存放接收到数据的缓冲区，参数 `len` 则是要接收的字节数。与发送操作不同的是有一个 `recv` 函数返回 0 的情况要处理，它表示在收到任何数据之前先收到了对端切断连接的通知。



11.2.14 多路复用

应用程序对文件进行读写操作时,可以使用阻塞方式或非阻塞方式。考虑对设备文件或 socket 进行读写操作的情况,这时阻塞是经常遇到的。当进程被阻塞时,就不可能再对其他文件做出响应。如果使用非阻塞式的 I/O 操作,则进程必须反复地轮询文件,因为无法判断文件何时才有可读的数据或可以进行写操作,这样做一般来说工作效率较低。

Linux 系统提供了一种多路复用的方式让单个进程能够同时监视多个文件描述符。实现多路复用的一种方式是使用 `select` 函数,其接口头文件与原型如下:

```
#include <sys/select.h>
int select(int nfd, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

其各个参数及返回值的含义如下。

- ◆ `nfd`: 所监视的文件描述符的最大值加 1。
- ◆ `readfds`: 表示可读状态的被监视的文件描述符集合。
- ◆ `writefds`: 表示可写状态的被监视的文件描述符集合。
- ◆ `exceptfds`: 表示异常状态的被监视的文件描述符集合。
- ◆ `timeout`: 表示超时的时间长度。如果对函数的调用阻塞了这个长度的时间仍未有任何被监视的文件描述符成为就绪状态,函数将返回。可以为 `NULL`,表示没有超时限制,函数将一直等待。
- ◆ 返回值: 所有文件描述符集合中处于就绪状态的个数,超时则返回 0,发生错误则返回 -1 并设置 `errno` 变量的值为错误码。

这里的就绪状态指的是读文件时不会阻塞,或者写文件时不会阻塞,或者文件有异常状态待处理。

使用 `select` 函数就可以实现同时监视多个 I/O 设备,只要其中任意一个达到所要求的状态(可读或可写),函数就会返回,返回之后三个文件描述符集合的内容将被修改,只留下那些处于就绪状态的文件描述符。对这些文件描述符可以进行相应的操作而不会阻塞。因此应用的关键就在于文件描述符集合的操作,系统为此定义了多个宏,形式如下:

```
FD_ZERO(fd_set *set);
FD_SET(int fd, fd_set *set);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
```

其中参数 `fd` 是一个文件描述符,参数 `set` 是要操作的文件描述符集合的指针,各个宏的作用如下。

- ◆ `FD_ZERO`: 将 `set` 指向的文件描述符集合初始化,即清空所有的文件描述符。
- ◆ `FD_SET`: 设置文件描述符 `fd` 到 `set` 指向的集合中。
- ◆ `FD_CLR`: 从 `set` 指向的集合中清除某个文件描述符。
- ◆ `FD_ISSET`: 测试文件描述符 `fd` 是否在 `set` 指向的集合中。

由于 `select` 函数返回时会对输入的文件集合进行修改，去掉那些不在就绪状态的文件描述符，因此如果再次调用 `select`，则必须重新设定这些集合的内容。

11.3 socket 编程实例

本节将通过多个例程来说明 `socket` 编程的一般思路。

11.3.1 TCP 与 UDP 程序流程

TCP 通信的一般流程如图 11.2 所示。

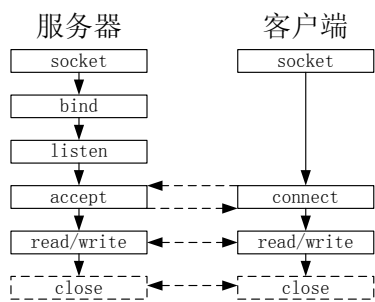


图 11.2 TCP 协议通信流程

TCP 通信采用服务器/客户端模型，通信双方的地位是不对等的。在服务器端需要有一个 `socket` 专门用于侦听并接受来自客户端的请求，接受成功后将会建立一个新的 `socket` 与客户端进行通信。而客户端只需要用一个 `socket` 向服务器端发起连接请求，连接成功后这个 `socket` 即可用于通信。在通信过程中，双方都可以主动中断连接，这将导致对方正在进行或随后进行的发送接收操作出错。

UDP 通信的一般流程如图 11.3 所示。

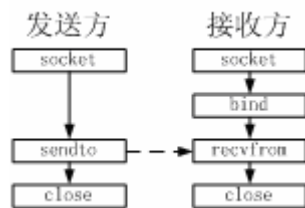


图 11.3 UDP 协议通信流程

在 UDP 通信过程中没有连接的概念，可以随时使用任何一个 `socket` 向对方发送数据。接收数据则需要先将 `socket` 与一个地址绑定。

11.3.2 TCP 通信例程

下面是一个使用 TCP 进行通信的例程。例程由两个文件组成：`tserver.c` 和 `tclient.c`，分别作为服务器端和客户端的程序。

文件 `tserver.c` 的源码如下：

```

/* 文件名: tserver.c */
/* 说明: TCP 通信例程服务器端程序 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* 输出错误信息并退出进程 */
void die(char *msg)
{
    perror(msg);
    exit(1);
}

/* 从文件 from 向 to 复制数据 */
void copy_data(int from, int to)
{
    char buf[1024];
    int amount;
    while ((amount = read(from, buf, sizeof(buf))) > 0) {
        if (write(to, buf, amount) != amount) {
            die("write");
            return;
        }
    }
    if (amount < 0) die("read");
}

/* 主函数 */
int main(void)
{
    int sock, conn, i;
    struct sockaddr_in addr;
    size_t addr_len = sizeof(addr);
    if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        die("socket"); /* 出错退出 */
    }
    /* 设置地址可重用 */
    i = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &i, sizeof(i));
    /* 绑定到任意地址 */
    memset(&addr, 0, sizeof(addr)); /* 将数据清 0 */
    addr.sin_family = AF_INET; /* 地址类型为 IPv4 */
    addr.sin_addr.s_addr = htonl(INADDR_ANY); /* IP 地址为任意 */
    addr.sin_port = htons(8888); /* 指定端口为 8888 */
    if (bind(sock, (struct sockaddr *)&addr, sizeof(addr))) {

```



```

        die("bind"); /* 出错退出 */
    }
    if (listen(sock, 5)) {
        die("listen"); /* 出错退出 */
    }
    while ((conn = accept(sock, (struct sockaddr *)&addr, &addr_len)) >= 0) {
        printf("Accept %s:%d\n", inet_ntoa(addr.sin_addr), addr.sin_port);
        /* 使用接受后的新 socket 通信 */
        copy_data(conn, STDOUT_FILENO); /* 读取数据并写入到标准输出 */
        printf("Done!\n");
        close(conn);
    }
    if (conn < 0) die("accept"); /* 出错退出 */
    close(sock);
    return 0;
}

```

文件 `tclient.c` 的源码如下:

```

/* 文件名: tclient.c */
/* 说明: TCP 通信例程客户端程序 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>

/* 定义与 tserver.c 文件中相同的 die 和 copy_data 函数 */

/* 主函数 */
int main(int argc, const char *argv[])
{
    struct sockaddr_in addr;
    int sock;
    if (argc != 2) { /* 需要 2 个命令行参数 */
        fprintf(stderr, "need an IP address\n");
        return 1;
    }
    memset(&addr, 0, sizeof(addr)); /* 清空数据 */
    addr.sin_family = AF_INET; /* 设置地址类型 */
    inet_aton(argv[1], &addr.sin_addr); /* 设置 IP 地址 */
    addr.sin_port = htons(8888); /* 设置端口号 */
    if ((sock = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
        die("socket");
    }
    if (connect(sock, (struct sockaddr *)&addr, sizeof(addr))) {
        die("connect");
    }
}

```




```

printf("Connected!\n");
/* 从标准输入读取数据写入 sock */
copy_data(STDIN_FILENO, sock);
close(sock);
return 0;
}

```

这个例程是使用 `read/write` 函数来操作 `socket` 的，客户端从标准输入读取数据并写入与服务器端通信的 `socket`，服务器端则从 `socket` 读取数据写入标准输出。运行的结果就是客户端输入的数据显示在服务器端的屏幕上。注意在运行客户端程序时，要求输入一个 IP 地址作为参数，它表示要连接的服务器的地址。

11.3.3 多进程并发服务器应用

前述 TCP 服务器端程序有一个缺陷，就是同时只能接受一个客户端请求，只有这个客户端通信结束后，其他客户端才可以连接到服务器进行通信。

显然，这个服务器缺乏并发通信的能力，在现实中意义并不大。现代的网络服务器应用要求具有并发通信的能力，也就是能够同时服务多个不同的客户应用，这一点可以由多进程来实现。当服务器接受一个新的客户后，创建一个新的进程，新进程使用新创建的 `socket` 与客户端通信，而原来的进程使用原来的 `socket` 继续接受连接请求。

下面以一个例程来说明如何编写多进程的服务器程序，程序源码如下：

```

/* 文件名: tcp_fork.c */
/* 说明: TCP 多进程服务器例程 */

#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

#define DEFAULT_PORT    8888    /* 默认端口 */
#define BACKLOG         5      /* 连接请求队列长度 */
#define BUF_SIZE        128    /* 缓冲区大小 */

static int send_str(int sock, const char *str);
static int recv_str(int sock, char *buf, int max_len);
static int send_byte(int sock, const char *buf, int len);
static int recv_byte(int sock, char *buf, int len);
static int eat_byte(int sock, int len);

static int start_server(int port);
static int start_client(const char *addr, int port);

```



```
static char g_buf[BUF_SIZE]; /* 缓冲区 */

/* 主函数 */
int main(int argc, char *argv[])
{
    int i;
    int r;
    int port;
    int server_mode;
    char *addr;
    port = DEFAULT_PORT;
    server_mode = 0;
    addr = "127.0.0.1";
    /* 对命令行参数进行分析 */
    for (i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-s") == 0) { /* 如果是 -s */
            server_mode = 1; /* 说明要以服务器方式启动 */
        } else if (strcmp(argv[i], "-p") == 0) { /* 如果是 -p */
            i++; /* 下一个参数 */
            if (i >= argc) {
                printf("Error: must specify port number!\n");
                return -1;
            }
            port = atoi(argv[i]); /* 是端口号 */
            if (port == 0) {
                printf("Error: wrong port number!\n");
                return -1;
            }
        } else { /* 否则 */
            addr = argv[i]; /* 是 IP 地址 */
        }
    }
    if (server_mode) {
        r = start_server(port); /* 启动服务器 */
    } else {
        r = start_client(addr, port); /* 启动客户端 */
    }
    return r;
}

/* 服务器程序 */
int start_server(int port)
{
    int r;
    int val;
    int sock_listen, sock;
    struct sockaddr_in host_addr, peer_addr;
    pid_t pid;
    struct sigaction sa;
    char prompt[25];
```



```

socklen_t socklen;
/* 打开 socket */
sock_listen = socket(PF_INET, SOCK_STREAM, 0);
if (sock_listen < 0) {
    perror("start_server: socket()");
    return -1;
}
/* 设置地址可重用 */
val = 1;
setsockopt(sock_listen, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
/* 绑定 */
memset(&host_addr, 0, sizeof(host_addr));
host_addr.sin_family = AF_INET;
host_addr.sin_addr.s_addr = htonl(INADDR_ANY);
host_addr.sin_port = htons(port);
r = bind(sock_listen, (struct sockaddr *)&host_addr, sizeof(host_addr));
if (r < 0) {
    perror("start_server: bind()");
    close(sock_listen);
    return -1;
}
/* 侦听 */
r = listen(sock_listen, BACKLOG);
if (r < 0) {
    perror("start_server: listen()");
    close(sock_listen);
    return -1;
}
printf("Server start listening on port %d.\n", port);
/* 设置信号处理方式以避免产生僵尸进程 */
memset(&sa, 0, sizeof(sa));
sa.sa_handler = SIG_DFL;
sa.sa_flags = SA_NOCLDWAIT;
sigaction(SIGCHLD, &sa, NULL);
/* 主循环 */
for ( ; ; ) {
    /* 接受 */
    socklen = sizeof(peer_addr);
    sock = accept(sock_listen, (struct sockaddr *)&peer_addr, &socklen);
    if (sock < 0) {
        perror("start_server: accept()");
        continue;
    }
    /* 接受成功则生成子进程 */
    pid = fork();
    if (pid != 0) { /* 主进程 */
        /* 这里应该将 sock 关闭, 因为主进程不再使用 */
        /* 不会切断连接, 因为子进程在使用这个 sock */
        close(sock);
        continue;
    }
}

```

```

        /* 保存对端 IP 地址和端口以备将来显示 */
        sprintf(prompt, "%s:%d",
            inet_ntoa(peer_addr.sin_addr),
            ntohs(peer_addr.sin_port));
        printf("Accept remote %s, pid = %d\n", prompt, getpid());
        /* 收发数据循环 */
        for ( ; ; ) {
            /* 发送字符串 OK 并接收来自客户端的数据 */
            if (send_str(sock, "OK") <= 0
                || recv_str(sock, g_buf, sizeof(g_buf)) <= 0) {
                printf("%s error or disconnected, "
                    "socket is closing...\n",
                    prompt);
                close(sock);
                return 0;
            }
            /* 输出来自客户端的数据 */
            printf("%s> %s\n", prompt, g_buf);
        }
    }
    return 0;
}

/* 启动客户端 */
int start_client(const char *addr, int port)
{
    int r;
    int sock;
    struct sockaddr_in server_addr;
    /* 打开 socket */
    sock = socket(PF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("start_client: socket()");
        return -1;
    }
    /* 连接 */
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    if (!inet_aton(addr, &server_addr.sin_addr)) {
        perror("start_client: inet_aton()");
        close(sock);
        return -1;
    }
    server_addr.sin_port = htons(port);
    r = connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr));
    if (r < 0) {
        perror("start_client: connect()");
        close(sock);
        return -1;
    }
    /* 接收与发送循环 */

```

```

    for ( ; ; ) {
        /* 接收一个字符串 */
        if (recv_str(sock, g_buf, sizeof(g_buf)) <= 0) break;
        /* 显示接收到的字符串 */
        printf("> %s\n", g_buf);
        /* 让用户输入一行数据 */
        fgets(g_buf, sizeof(g_buf), stdin);
        g_buf[strlen(g_buf)-1] = '\0';
        /* 将数据发送给服务器 */
        if (send_str(sock, g_buf) <= 0) break;
    }
    close(sock);
    return 0;
}

/* 发送字符串 */
int send_str(int sock, const char *str)
{
    int byte;
    int len;
    int len_send;
    /* 首先发送一个整数, 表示字符串的长度 */
    len = strlen(str);
    len_send = htonl(len);
    byte = send_byte(sock, (char *)&len_send, sizeof(len_send));
    if (byte < 0) return -1;
    /* 然后发送字符串本身 */
    byte = send_byte(sock, str, len);
    if (byte < 0) return -1;
    return byte;
}

/* 接收字符串 */
int recv_str(int sock, char *buf, int max_len)
{
    int byte;
    int len;
    int len_recv;
    int len_diff;
    len_diff = 0;
    /* 首先接收一个整数, 这个整数表示字符串长度 */
    byte = recv_byte(sock, (char *)&len_recv, sizeof(len_recv));
    if (byte < 0) return -1;
    len = ntohl(len_recv);
    if (len > max_len) {
        len_diff = len-max_len;
        len = max_len;
    }
    /* 然后接收字符串 */
    byte = recv_byte(sock, buf, len);
    if (byte <= 0) return -1;

```



```
    buf[byte] = '\0';
    /* 如果实际接收的字符数小于字符串的长度，则消耗掉多余的字符 */
    if (len_diff > 0) eat_byte(sock, len_diff);
    return byte;
}

/* 可靠发送 len 个字节 */
int send_byte(int sock, const char *buf, int len)
{
    int rc;
    int byte;
    for (byte = 0; byte < len; byte += rc) {
        rc = send(sock, buf+byte, len-byte, MSG_NOSIGNAL);
        if (rc < 0 && errno != EINTR) {
            byte = -1;
            break;
        }
    }
    return byte;
}

/* 可靠接收 len 个字节 */
int recv_byte(int sock, char *buf, int len)
{
    int rc;
    int byte;
    for (byte = 0; byte < len; byte += rc) {
        rc = recv(sock, buf+byte, len-byte, MSG_NOSIGNAL);
        if (rc == 0) break;
        if (rc < 0 && errno != EINTR) {
            byte = -1;
            break;
        }
    }
    return byte;
}

/* 消耗 len 个字节 */
int eat_byte(int sock, int len)
{
    int rc;
    int byte;
    char buf[32];
    for (byte = 0; len-byte > 32; byte += rc) {
        rc = recv(sock, buf, 32, MSG_NOSIGNAL);
        if (rc == 0) break;
        if (rc < 0 && errno != EINTR) return -1;
    }
    if (rc != 0) byte += recv_byte(sock, buf, len-byte);
    return byte;
}
```



在这个例程中，使用了自定义的协议来传输一个字符串，即先发送 4 个字节，表示字符串的长度，然后再发送字符串本身。这个协议是必要的，因为在数据流形式的传输中，接收端无法分辨字符串在哪里结束。程序采用了根据命令行参数来决定启动服务器还是客户端的方式，将两者合为一个源码，这样很多函数就可以由两者同时使用。

假设编译后可执行程序名为 `tcp_fork`，则必须用下面的命令启动服务器进程：

```
./tcp_fork -s -p 1234
```

这里 `-p` 用于指定端口号，可省略，默认是代码中所写的 8888。

对应的客户端进程则应该用如下命令启动：

```
./tcp_fork -p 1234 127.0.0.1
```

这里 127.0.0.1 是要连接的主机 IP 地址，这个 IP 地址是主机自己的 IP 地址。

11.3.4 多路复用服务器应用

使用多进程并发的服务器应用可以解决同时接受多个客户端连接的问题，但也有其缺点。首先在进程之间需要共享数据的情况下，多进程可能会有复杂的同步问题，其次创建新进程本身要消耗一定的系统资源，进程间的频繁切换也会使执行效率降低。在嵌入式领域，主机的各种资源都比较有限，这种方案的缺点就有可能突显出来。

使用多路复用的方式可以同时监视多个文件描述符，这样，不需要多进程就可以对多个 socket 描述符做出响应，从而接受多个客户端的同时连接。这种方式下需要注意的问题是，服务器程序在处理某个客户端的连接请求或通信时不能阻塞，否则将暂时失去对其他客户端的响应。

这里将前述多进程的服务器程序改造成使用多路复用来实现，其源码如下：

```
/* 文件名: tcp_select.c */
/* 说明: 使用多路复用的服务器例程 */

#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>

#define DEFAULT_PORT    8888    /* 默认端口 */
#define BACKLOG         5      /* 连接请求队列长度 */
#define BUF_SIZE        128    /* 缓冲区大小 */
#define CLIENT_NUM      128    /* 允许同时连接的客户端最大个数 */

/* 用于保存一个客户端的信息 */
struct client {
    int sockfd; /* socket 描述符 */
```



```
char buf[BUF_SIZE+1]; /* 缓冲区 */
int tail; /* 指向缓冲区结尾 */
char ip[16]; /* IP 地址 (字符串) */
unsigned short port; /* 端口号 */
};

static int send_str(int sock, const char *str);
static int recv_str(int sock, char *buf, int max_len);
static int send_byte(int sock, const char *buf, int len);
static int recv_byte(int sock, char *buf, int len);
static int eat_byte(int sock, int len);

static int accept_client(int sock_listen);
static int process_client(int index);

static int start_server(int port);
static int start_client(const char *addr, int port);

static struct client client[CLIENT_NUM];
static char g_buf[BUF_SIZE];

/* 主模块 */
int main(int argc, char *argv[])
{
    int i;
    int r;
    int port;
    int server_mode;
    char *addr;
    port = DEFAULT_PORT;
    server_mode = 0;
    addr = "127.0.0.1";
    /* 对命令行参数进行分析 */
    for (i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-s") == 0) { /* 如果是 -s */
            server_mode = 1; /* 说明要以服务器方式启动 */
        } else if (strcmp(argv[i], "-p") == 0) { /* 如果是 -p */
            i++; /* 下一个参数 */
            if (i >= argc) {
                printf("Error: must specify port number!\n");
                return -1;
            }
            port = atoi(argv[i]); /* 是端口号 */
            if (port == 0) {
                printf("Error: wrong port number!\n");
                return -1;
            }
        } else { /* 否则 */
            addr = argv[i]; /* 是 IP 地址 */
        }
    }
}
```




```

    if (server_mode) {
        r = start_server(port); /* 启动服务器 */
    } else {
        r = start_client(addr, port); /* 启动客户端 */
    }
    return r;
}

/* 服务器程序 */
int start_server(int port)
{
    int i;
    int r;
    int val;
    int sock_listen;
    struct sockaddr_in host_addr;
    fd_set rfd;
    int nfd;
    /* 打开 socket */
    sock_listen = socket(PF_INET, SOCK_STREAM, 0);
    if (sock_listen < 0) {
        perror("start_server: socket()");
        return -1;
    }
    /* 设置地址可重用 */
    val = 1;
    setsockopt(sock_listen, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
    /* 绑定 */
    memset(&host_addr, 0, sizeof(host_addr));
    host_addr.sin_family = AF_INET;
    host_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    host_addr.sin_port = htons(port);
    r = bind(sock_listen, (struct sockaddr *)&host_addr, sizeof(host_addr));
    if (r < 0) {
        perror("start_server: bind()");
        close(sock_listen);
        return -1;
    }
    /* 侦听 */
    r = listen(sock_listen, BACKLOG);
    if (r < 0) {
        perror("start_server: listen()");
        close(sock_listen);
        return -1;
    }
    printf("Server start listening on port %d.\n", port);
    /* 初始化客户端数据 */
    for (i = 0; i < CLIENT_NUM; i++) {
        client[i].sockfd = -1;
        client[i].tail = 0;
    }
}

```

```

/* 进入主循环 */
for ( ; ; ) {
    /* 设置文件集合 */
    FD_ZERO(&rfdset);
    FD_SET(sock_listen, &rfdset);
    nfds = sock_listen+1;
    for (i = 0; i < CLIENT_NUM; i++) {
        if (client[i].sockfd > 0) {
            FD_SET(client[i].sockfd, &rfdset);
            if (client[i].sockfd >= nfds) nfds = client[i].sockfd+1;
        }
    }
    /* 监视文件描述符 */
    if (select(nfds, &rfdset, NULL, NULL, NULL)) {
        /* 如果侦听的 socket 可读则进行接受操作 */
        if (FD_ISSET(sock_listen, &rfdset)) accept_client(sock_listen);
        /* 逐个检查对应各个客户端的 socket 是否就绪 */
        for (i = 0; i < CLIENT_NUM; i++) {
            if (client[i].sockfd < 0) continue;
            /* 如果就绪则进行处理 */
            if (FD_ISSET(client[i].sockfd, &rfdset)) process_client(i);
        }
    } else {
        perror("start_server: select()");
    }
}
return 0;
}

/* 接受客户端的连接请求 */
int accept_client(int sock_listen)
{
    int i;
    socklen_t socklen;
    struct sockaddr_in peer_addr;
    int sock;
    /* 接受 */
    socklen = sizeof(peer_addr);
    sock = accept(sock_listen, (struct sockaddr *)&peer_addr, &socklen);
    if (sock < 0) {
        perror("start_server: accept()");
        return -1;
    }
    /* 在客户端数据的数组中查找一个未用的, 并将新客户端的数据填入 */
    for (i = 0; i < CLIENT_NUM; i++) {
        if (client[i].sockfd > 0) continue;
        client[i].sockfd = sock;
        strcpy(client[i].ip, inet_ntoa(peer_addr.sin_addr));
        client[i].port = ntohs(peer_addr.sin_port);
        break;
    }
}

```

```

/* 如果找不到,说明能接受的客户端数目已达最大值 */
if (i == CLIENT_NUM) {
    printf("Cannot accept more client!\n");
    close(sock);
    return -1;
}
printf("Accept remote %s:%d\n", client[i].ip, client[i].port);
/* 发送字符串 OK */
send_str(sock, "OK");
return 0;
}

/* 处理客户端的通信 */
int process_client(int index)
{
    int byte;
    int sock = client[index].sockfd;
    char *buf = client[index].buf;
    int tail = client[index].tail;
    /* 接收数据 */
    byte = recv(sock, buf+tail, BUF_SIZE-tail, MSG_DONTWAIT);
    if (byte == 0 || (byte < 0 && errno != EINTR)) {
        printf("%s:%d error or disconnected, socket is closing...\n",
            client[index].ip, client[index].port);
        close(sock);
        client[index].sockfd = -1;
        client[index].tail = 0;
        return -1;
    }
    /* 移动缓冲区尾部指针 */
    tail += byte;
    if (tail > sizeof(int)) { /* 如果收到的数据超出一个整数的大小 */
        /* 解析这个整数,它表示字符串的长度 */
        int len = ntohl(*(int *)buf);
        if (tail-sizeof(int) == len) { /* 字符串已接收完全 */
            /* 显示字符串 */
            buf[tail] = '\0';
            printf("%s:%d > %s\n",
                client[index].ip, client[index].port, buf+sizeof(int));
            /* 回送 OK */
            send_str(sock, "OK");
            /* 移动缓冲区指针到开始 */
            tail = 0;
        } else if (tail == BUF_SIZE) { /* 字符串接收未完全但缓冲区已满 */
            /* 显示字符串 */
            buf[tail] = '\0';
            printf("%s:%d > %s\n",
                client[index].ip, client[index].port, buf+sizeof(int));
            /* 修改缓冲区中的字符串长度,并移动缓冲区指针 */
            /* 将字符串的剩余部分作为下一个字符串处理 */
            /* 因为不允许阻塞,所以不能再 eat_byte 消耗掉剩余字符 */

```

```

        *(int *)buf = htonl(len-BUF_SIZE+sizeof(int));
        tail = sizeof(int);
    }
}
client[index].tail = tail;
return 0;
}

/* 以下这些函数仅在客户端中使用 */
/* 定义与 tcp_fork 例程中相同的 start_client 函数 */
/* 定义与 tcp_fork 例程中相同的 send_str, recv_str 函数 */
/* 定义与 tcp_fork 例程中相同的 send_byte, recv_byte, eat_byte 函数 */

```

这个例程中，由于是单进程，所以必须针对每个客户端分别保存数据，包括所用的 `socket` 描述符、接收缓冲区等。另外，`select` 函数返回之后，对于处于就绪状态的 `socket` 不能再阻塞式的操作。

11.3.5 UDP 服务器应用

UDP 的通信模型本身没有服务器和客户端的概念，通信的双方地位完全是等同的。但实际上我们仍把等待接收数据并及时做出响应的一方称为服务器，而主动发送数据并等待对方响应的一方称为客户端。

这里我们将实现一个简单的 TFTP 服务器。它虽然并不完善，但已经可以用来给目标机下载文件了。例程的源码如下：

```

/* 文件名: stftp.c */
/* 说明: 简单的 TFTP 服务器例程 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/* TFTP 协议操作码定义 */
#define RRQ    1 /* 读数据请求 */
#define WRQ    2 /* 写数据请求 */
#define DATA  3 /* 数据块 */
#define ACK    4 /* 确认 */
#define ERROR  5 /* 发生错误 */

/* TFTP 协议错误码定义 */
#define FILE_NOT_FOUND 1 /* 文件未找到 */
#define BAD_OP          4 /* 非法操作 */

/* 显示错误信息并退出 */
void die(char * msg)
{

```

```

    perror(msg);
    exit(1);
}

/* TFTP 协议数据包格式 */
struct tftp_packet {
    short opcode; /* 2 字节操作码 */
    union {
        char bytes[514]; /* 最多 514 个字节, 2 字节块号 + 512 字节数据 */
        struct {
            short code; /* 错误码 */
            char message[200]; /* 错误信息 */
        } error; /* 错误包 */
        struct {
            short block; /* 块号 */
            char bytes[512]; /* 文件数据 */
        } data; /* 数据包 */
        struct {
            short block; /* 块号 */
        } ack; /* 确认包 */
    } u;
};

/* 发送错误包 */
void send_error(int s, struct sockaddr_in *addr, socklen_t addr_len, int code)
{
    struct tftp_packet err;
    int size;
    int bytes;
    err.opcode = htons(ERROR);
    err.u.error.code = htons(code);
    switch (code) {
        case FILE_NOT_FOUND:
            strcpy(err.u.error.message, "file not found");
            break;
        case BAD_OP:
            strcpy(err.u.error.message, "bad op");
            break;
        default:
            strcpy(err.u.error.message, "undefined");
    }
    /* 2 字节操作码 + 2 字节错误码 + 错误信息 + 字符串结束符 */
    size = 2 + 2 + strlen(err.u.error.message) + 1;
    /* 发送 */
    bytes = sendto(s, &err, size, 0,
        (struct sockaddr *)addr, addr_len);
    if (bytes != size) perror("send_error: sendto");
}

/* 处理请求 */
void handle_request(int sock, struct sockaddr_in *from, socklen_t from_len,
    struct tftp_packet *request) {
    char *file_name;
    char *mode;
    int fd;

```



```

int size;
int bytes;
struct tftp_packet data, response;
int block;
struct sockaddr_in new_from;
socklen_t new_from_len;
/* 分析操作码 */
request->opcode = ntohs(request->opcode);
if (request->opcode != RRQ) return; /* 如果不是读请求则不处理 */
file_name = request->u.bytes; /* 得到文件名 */
mode = file_name + strlen(file_name) + 1; /* 得到传输模式 */
printf("Requested file name is %s, mode is %s\n", file_name, mode);
/* 我们只支持 octet 模式 */
if (strcmp(mode, "octet")) {
    send_error(sock, from, from_len, BAD_OP);
    return;
}
/* 打开文件 */
fd = open(file_name, O_RDONLY);
if (fd < 0) {
    send_error(sock, from, from_len, FILE_NOT_FOUND);
    return;
}
/* 准备发送数据 */
printf("Sending start.\n");
data.opcode = htons(DATA); /* 操作码为数据块 */
for (block = 1; (size = read(fd, data.u.data.bytes, 512)) > 0; block++) {
    data.u.data.block = htons(block);
    /* 加上 2 字节操作码, 2 字节块号的长度 */
    size += 4;
    /* 发送 */
    bytes = sendto(sock, &data, size, 0,
        (struct sockaddr *)&from, from_len);
    if (bytes != size) {
        perror("handle_request: sendto");
        close(fd);
        return;
    }
    /* 接收确认 */
    new_from_len = sizeof(new_from);
    bytes = recvfrom(sock, &response, sizeof(response), 0,
        (struct sockaddr *)&new_from, &new_from_len);
    if (bytes < 0) {
        perror("handle_request: recvfrom");
        close(fd);
        return;
    }
    /* 如果数据包非来自原来的客户端, 说明受到干扰 */
    if (new_from.sin_addr.s_addr != from->sin_addr.s_addr) {
        fprintf(stderr, "handle_request: bad address\n");
        close(fd);
        return;
    }
    /* 判断是否为确认 */
    response.opcode = ntohs(response.opcode);

```



```

        if (response.opcode != ACK) {
            fprintf(stderr, "handle_request: not ACK\n");
            close(fd);
            return;
        }
        /* 判断确认的块号是否正确 */
        response.u.ack.block = ntohs(response.u.ack.block);
        if (response.u.ack.block != block) {
            fprintf(stderr, "handle_request: wrong block\n");
            close(fd);
            return;
        }
        printf("."); /* 输出信息表示一块数据传输成功 */
    }
    printf("\nSent out.\n");
}

/* 主函数 */
int main(int argc, char ** argv)
{
    int s;
    int bytes;
    struct sockaddr_in addr, from;
    socklen_t from_len;
    struct tftp_packet packet;
    if ((s = socket(PF_INET, SOCK_DGRAM, 0)) < 0) die("socket");
    /* 绑定 */
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(69); /* TFTP 服务默认端口号 */
    if (bind(s, (struct sockaddr *)&addr, sizeof(addr))) die("bind");
    /* 主循环 */
    for (;;) {
        /* 接收 */
        from_len = sizeof(from);
        bytes = recvfrom(s, &packet, sizeof(packet), 0,
            (struct sockaddr *)&from, &from_len);
        if (bytes < 0) die("recvfrom");
        printf("Request from %s:%d\n",
            inet_ntoa(from.sin_addr), htons(from.sin_port));
        /* 处理请求 */
        handle_request(s, &from, from_len, &packet);
    }
}

```

在例程中采用了最简单的方式,即同时只能有一个客户端进行文件传输,如果同时有其他客户端请求传输文件,则正常的传输过程会被干扰而中断。

需要说明的是, Linux 系统中认为 1024 以下的端口号比较重要,因此需要 root 权限才能成功绑定。上述例程要绑定 TFTP 服务的默认端口号 69,因此需要 root 权限才能执行成功。

第 12 章 多线程并发程序设计

多线程是现代操作系统所支持的一种重要的多任务机制。现在的 Linux 系统已经在内核级别真正实现了对多线程的支持，线程也成为了内核进行调度的基本单元。每个用户级的线程都一一映射到内核，一个线程进行系统调用被阻塞不会影响同一进程所创建的其他线程。在应用接口上，可以使用 NPTL (Native POSIX Thread Library) 线程库，它基本上实现了 POSIX 标准所定义的线程模型。

本章的主要内容是介绍 Linux 上的多线程编程以及多线程之间的同步方式。

12.1 线程的概念

为了理解线程，可以先回顾一下进程的概念。进程包括程序映像、地址空间等要素。内核使用 PCB 来管理进程。进程是内核进行调度的基本单元，每个独立的进程都有自己的代码段、数据段及堆栈，它们使用自己的虚拟地址空间，多个进程间互不影响。

实际上，在 Linux 内核中，线程与进程是以统一的方式来管理的。线程也是内核进行调度的基本单元。不同点在于，线程没有自己独立的地址空间，创建出来的新线程将和创建它的进程（或线程）共享同一个虚拟地址空间，它们被置于同一个线程组中。线程的代码段、全局数据段、堆及其他一些诸如文件描述符表等内核管理的东西在同一组内是共享的，但是每个线程都有自己独立的栈空间。因此，多线程环境下对全局变量的访问有可能需要同步，但是局部变量仍然有各自的存储空间，互不影响。

与多进程相比，多线程具有以下优点。

一、线程适合多任务通信应用环境

线程类似于进程。如同进程一样，线程由内核进行调度。在单处理器系统中，内核使用按时间片轮流执行的方式来模拟线程的并发执行；在多处理器系统中，多个线程也可以真正地同时执行。

由于同一进程中创建的多个线程共享相同的内存空间，不同的线程可以存取内存中的同一个变量，因此实现多线程间的通信比多进程间的通信更简单方便。这种直接共享变量的方式省去了进行 IPC 系统调用的开销，执行效率更高，但是需要注意同步问题。

二、线程的创建和调度更高效

在创建线程时，内核无须单独复制进程的内存空间或文件描述符等，这就节省了大量的 CPU 时间，因而比创建进程快很多。线程切换时不需要更换整个页表，因此效率也会有所提高。

12.2 线程编程接口

在 Linux 系统上可以使用 clone 系统调用创建新的线程，但它是 Linux 上特有的系统调用，因此不利于跨平台的移植。通常进行多线程编程时，我们使用的是 POSIX 标准所规定的 API 接口，

这时可以不关心具体平台对线程的实现，使程序获得较好的可移植性。

本节的主要内容是介绍 POSIX 线程的编程接口，要使用这些接口，必须包含以下头文件：

```
#include <pthread.h>
```

在编译线程相关的程序时也要注意，因为 POSIX 线程接口不是 C 标准库的一部分，所以必须与 `libpthread` 或 `librt` 共享库链接。

12.2.1 线程的创建与退出

使用 `pthread_create` 函数可以创建新线程，其原型如下：

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void*), void *arg);
```

其各个参数及返回值的含义如下。

- ◆ `thread`：指向一个 `pthread_t` 型变量，用于返回线程句柄。
- ◆ `attr`：指向描述新线程属性的数据，可以为 `NULL`，表示默认属性。
- ◆ `start_routine`：新线程的工作函数。
- ◆ `arg`：传递给执行函数的参数。
- ◆ 返回值：0 表示成功，非 0 表示有错误发生。

类似于进程的 PID，线程由一个 `pthread_t` 型的数据代表，可称为线程句柄。创建线程成功以后，`pthread_create` 函数将线程句柄放在 `thread` 参数指向的变量中返回。

与创建进程不同，创建线程时可以指定一个工作函数，新线程将从这个函数开始执行，函数返回也就等价于线程退出。工作函数必须有一个 `(void *)` 型的参数，新线程开始执行时，这个参数的值就是 `pthread_create` 函数的 `arg` 参数的值，因此可以利用它来向线程传递数据。工作函数还必须有 `(void *)` 型的返回值，它代表线程的退出状态。

类似于 `exit` 函数，在线程中调用下面的函数可以退出线程：

```
void pthread_exit(void *value_ptr);
```

其中 `value_ptr` 参数将作为线程的退出状态。这个函数不会返回。

与 `waitpid` 函数类似，可以用下面的函数等待一个线程运行结束：

```
int pthread_join(pthread_t thread, void **value_ptr);
```

其中 `thread` 参数指定要等待的线程，`value_ptr` 是指向 `(void *)` 型数据的指针，用于获得线程的退出状态，如果不需要退出状态则将其设为 `NULL`。这个函数返回 0 表示执行成功，非 0 则表示执行失败。

12.2.2 线程属性

在创建线程时可以用一个 `pthread_attr_t` 型数据指定线程的属性。POSIX 标准中将其定义为线程属性对象而不是简单的结构体，并且提供了一些操作线程属性对象的函数。这些函数与线程属

性对象的具体实现无关，使用时不必关心 `pthread_attr_t` 类型的具体定义，它们的原型如下：

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getstacksize(const pthread_attr_t *attr,
                               size_t *stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr,
                               size_t stacksize);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedparam(const pthread_attr_t *attr,
                                struct sched_param *param);
int pthread_attr_setschedparam(pthread_attr_t *attr,
                                const struct sched_param *param);
```

`pthread_attr_init` 函数用来初始化一个线程属性对象，也就是将它设为默认值。`pthread_attr_destroy` 函数用来销毁一个线程属性对象，实际上就是将它设为一个非法的值。这两个函数返回 0 表示操作成功，非 0 表示操作失败。

`pthread_attr_getdetachstate` 函数用于获得线程属性对象的分离状态，所得到的分离状态将放在 `detachstate` 参数指向的整型变量中返回。`pthread_attr_setdetachstate` 函数则用于设置线程属性对象的分离状态为 `detachstate` 参数指定的值。分离状态的取值可以是 `PTHREAD_CREATE_DETACHED` 或 `PTHREAD_CREATE_JOINABLE`。这两个函数返回 0 表示操作成功，非 0 表示操作失败。

`pthread_attr_getstacksize` 函数用于获得线程属性对象的栈大小，所得到的大小放在 `stacksize` 参数指向的变量中返回。`pthread_attr_setstacksize` 函数则可以将线程属性对象的栈大小设为参数 `stacksize` 指定的值。这两个函数返回 0 表示操作成功，非 0 表示操作失败。

`pthread_attr_getschedpolicy` 函数用于获得线程属性对象的调度策略，所得到的调度策略放在 `policy` 参数指向的变量中返回。`pthread_attr_setschedpolicy` 函数则可以将线程属性对象的调度策略设为参数 `policy` 指定的值。这里的调度策略有以下几个取值：`SCHED_FIFO`、`SCHED_RR` 以及 `SCHED_OTHER`。这两个函数返回 0 表示操作成功，非 0 表示操作失败。

`pthread_attr_getschedparam` 函数用于获得线程属性对象的调度参数，所得到的调度参数放在 `param` 参数指向的变量中返回。`pthread_attr_setschedparam` 函数则可以将线程属性对象的调度参数设为参数 `param` 指定的值。当调度策略为 `SCHED_FIFO` 和 `SCHED_RR` 时，表示调度参数的数据类型 `struct sched_param` 只有一个成员 `sched_priority` 是有效的，表示调度的优先级。这两个函数返回 0 表示操作成功，非 0 表示操作失败。

使用线程属性时，首先定义一个线程属性对象，然后将其初始化，再用其他函数修改它的值，最后再用这个属性对象去创建线程，示例代码如下：

```
pthread_attr_t attr;
pthread_t tid;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
```

```
pthread_create(&tid, &attr, work_func, NULL);
```

12.2.3 线程的分离状态

当创建线程时所用的线程属性对象的分离状态是 `PTHREAD_CREATE_JOINABLE` 时, 线程运行结束后不会自动清理所占的资源, 直到对它调用了 `pthread_join` 函数。在这期间, 线程虽然运行结束, 但是用于管理它的数据仍然存在, 类似于僵尸进程的概念。

如果创建线程时所用的线程属性对象的分离状态是 `PTHREAD_CREATE_DETACHED`, 我们称线程处于分离状态。这种状态下的线程结束后会自动清理所占用的资源, 如果对它调用 `pthread_join` 函数就会返回错误。

默认情况下创建的线程是处于未分离状态的, 在创建以后可以用以下函数将其设为分离状态:

```
int pthread_detach(pthread_t thread);
```

这个函数返回 0 表示操作成功, 非 0 表示操作失败。

`pthread_detach` 函数不会导致调用者阻塞, 也不会导致所操作的线程结束。如果调用 `pthread_detach` 函数时线程已经结束, 则清理其所占用的资源。

综上所述, 对于创建时处于未分离状态的线程, 必须调用一次 `pthread_join` 函数或 `pthread_detach` 函数, 否则线程结束后就会留下没有释放的资源。

12.2.4 线程应用实例

下面通过一个简单的例程来说明线程编程接口的使用, 代码如下:

```
/* 文件名: thread_sum.c */
/* 说明: 使用线程求两个整数的和 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

/* 包装两个整数 */
typedef struct {
    int n[2];
} pairint;

/* 线程的工作函数 */
void *sum_thread(void *arg)
{
    printf("sum thread started!\n");
    pairint *p = (pairint *)arg; /* 取得指向两个整数的指针 */
    printf("thread id: %x\n", (int)pthread_self());
    printf("Sum of %d and %d is %d\n", p->n[0], p->n[1], p->n[0]+p->n[1]);
    sleep(3);
    return "sum thread result";
}
```



```
int main(void)
{
    pairint *p;
    void *thread_result;
    pthread_t threadid;
    p = malloc(sizeof(pairint)); /* 分配空间以存放两个整数 */
    p->n[0] = 8;
    p->n[1] = 9;
    /* 创建线程 */
    pthread_create(&threadid, NULL, sum_thread, (void *)p);
    pthread_join(threadid, &thread_result);
    printf("thread exit, result is %s\n", (char *)thread_result);
    /* 线程已结束, 所用空间可安全释放 */
    free(p);
    return 0;
}
```

这个例程使用线程来求两个整数的和。使用线程时首先声明了一个类型为 `pthread_t` 型的变量 `threadid`, 用来存放线程的句柄, 然后调用 `pthread_create` 函数创建线程, 同时也就得到了线程的句柄。

线程的工作函数是 `sum_thread`, 它把参数认为是指向 `pairint` 型变量的指针, 并从这个指针指向的内存中取出两个整数进行相加。因此, 主函数在创建线程时必须将指向 `pairint` 型变量的指针传递给线程, 并且这个指针指向的变量必须在线程运行期间一直存在, 这也是为什么使用动态分配内存而不是局部变量的原因。当然, 在这个例子中使用局部变量也没有问题, 因为在主函数返回前线程已经结束。但在复杂的程序中, 局部变量将在函数返回时消失, 如果线程仍然持续运行, 则有可能访问到非法内存, 这也是线程编程中特别需要注意的问题。

`pthread_self` 函数用于在线程中得到自身的句柄。

`pthread_create` 函数成功返回之后, 这个进程中将会有两个线程, 一个是我们创建的新线程, 另一个实质上是原来的进程, 习惯上称之为主线程。这里进程实际上代表的是一个线程组。创建线程后, 主线程和新线程到底哪一个先被调度运行是不确定的, 由内核的调度器决定。

创建新线程后主线程用 `pthread_join` 等待新线程的结束, 以确保进一步操作前新线程已结束。这里线程退出时的状态是一个指针值, 它指向一个字符串常数。由于这种字符串常数实际上在进程的数据段中, 并不随线程的退出而消失, 因此在主线程中访问它不会造成非法内存访问。

线程与进程的一个重要区别是: 线程没有像进程那样的父子关系, 仅有属于同一个进程的“同组”关系。在 POSIX 线程模型中, 主线程可以创建一个新线程 A, 新线程 A 又可以创建另一个新线程 B, 线程 A 和 B 本身没有父子关系, 只是同属于一个进程。等待线程结束的 `pthread_join` 操作可以由任何一个同组的线程发起, 不必是主线程。另外, 如果主线程退出, 即进程退出, 则所有的线程也会随之退出。

12.3 线程的同步

与多进程一样, 多个线程也是并发执行的, 并且一个进程中的所有线程都能够访问这个进程的全局变量, 因此线程同步就成了多线程环境下编程的一个核心问题。



POSIX 线程的同步手段除了我们熟知的信号灯外，还有互斥体、条件变量等。本节主要讲述各种线程同步手段的使用方法。

12.3.1 使用信号灯进行线程同步

POSIX 信号灯既可以用于进程间的同步，也可以用于线程间的同步。信号灯用于线程间的同步时更简单，只需将信号灯定义为全局变量并使用无名信号灯即可，因为同一进程内的各个线程自然共享所有的全局变量。

12.3.1.1 哲学家吃饭问题

下面以一个例程来说明信号灯在线程中的使用。这个例子是著名的“哲学家吃饭问题”。问题的描述是这样的：有若干个哲学家围坐在一个圆桌旁，每两人中间有一根筷子。哲学家面前的盘子中有若干食物，一个哲学家必须同时拿到他旁边的两根筷子才能吃一次饭，然后必须将筷子放下以便他人使用，要求模拟这一过程。

例程的代码如下：

```
/* 文件名: ph-eat.c */
/* 说明: 使用线程模拟哲学家吃饭问题 */

#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define NUM      5 /* 哲学家的个数 */
#define FOOD     5 /* 食物的个数 */

#define LEFT(x) (x) /* 得到左边的筷子编号 */
#define RIGHT(x) ((x)+1)%NUM /* 得到右边的筷子编号 */

static sem_t sem[NUM]; /* 表示筷子 */
static int food[NUM]; /* 剩余食物数量 */
static int stick[NUM]; /* 记录筷子被哪个哲学家拿走 */

/* 输出剩余食物的情况 */
void print_food()
{
    int i;
    printf("\033[1;31m Food: ");
    for (i = 0; i < NUM; i++) printf(" %1d ", food[i]);
    printf("\033[m\n");
}

/* 输出筷子的使用情况 */
void print_stick()
{
    int i;
```

```

    printf("Stick: ");
    for (i = 0; i < NUM; i++) {
        if (stick[i] < 0) printf(" - "); else printf(" %ld ", stick[i]);
    }
    printf("\n");
}

/* 用于模拟一个哲学家的线程工作函数, arg 参数代表哲学家的编号 */
void *th_eat(void *arg)
{
    int id = (int)arg;
    printf("th_eat started! ID = %d.\n", id);
    while (food[id] > 0) {
        sleep(1);
        sem_wait(&sem[LEFT(id)]); /* 等待拿左边的筷子 */
        stick[LEFT(id)] = id;
        print_stick();
#ifdef 0
        sleep(1);
#endif
        sem_wait(&sem[RIGHT(id)]); /* 等待拿右边的筷子 */
        stick[RIGHT(id)] = id;
        print_stick();
        food[id]--; /* 食物的数量减 1 */
        print_food();
        sleep(1);
        stick[LEFT(id)] = -1;
        print_stick();
        sem_post(&sem[LEFT(id)]); /* 放下左边的筷子 */
        stick[RIGHT(id)] = -1;
        print_stick();
        sem_post(&sem[RIGHT(id)]); /* 放下右边的筷子 */
    }
    return NULL;
}

int main(void)
{
    int i;
    pthread_t th[NUM];
    /* 初始化信号灯、食物数量和筷子的使用情况 */
    for (i = 0; i < NUM; i++) {
        sem_init(&sem[i], 0, 1);
        food[i] = FOOD;
        stick[i] = -1;
    }
    /* 生成 NUM 个线程, 代表 NUM 个哲学家 */
    for (i = 0; i < NUM; i++) {
        if (pthread_create(&th[i], NULL, th_eat, (void *)i)) {
            perror("main: pthread_create");
            return -1;
        }
    }
}

```

```

    }
}
/* 等待所有线程结束 */
for (i = 0; i < NUM; i++) pthread_join(th[i], NULL);
/* 销毁所有信号灯 */
for (i = 0; i < NUM; i++) sem_destroy(&sem[i]);
return 0;
}

```

在上述例程中，我们用信号灯来代表筷子，信号灯的值 1 表示筷子未使用，为 0 表示筷子在使用中，于是拿起筷子就等于获取信号灯，放下筷子就等于释放信号灯。然后使用 NUM 个线程来模拟 NUM 个哲学家的行为。

程序的运行结果如下：

```

th_eat started! ID = 1.
th_eat started! ID = 0.
th_eat started! ID = 2.
th_eat started! ID = 3.
th_eat started! ID = 4.
Stick:  - 1  -  -  -
Stick:  - 1  1  -  -
  Food:  5  4  5  5  5
Stick:  0  1  1  -  -
Stick:  0  1  1  3  -
Stick:  0  1  1  3  3
  Food:  5  4  5  4  5
Stick:  0  -  1  3  3
Stick:  0  -  -  3  3
Stick:  0  0  -  3  3
  Food:  4  4  5  4  5

```

由于输出的信息较多，这里只列出了开始的几行。必须指出的是，这些线程启动的顺序是不确定的，因此每次运行的输出结果也不尽相同。

上述例程中一个有趣的现象就是死锁的发生，如果将例程中被 `#if 0` 注释掉的部分启用，则例程的运行结果如下：

```

th_eat started! ID = 1.
th_eat started! ID = 2.
th_eat started! ID = 3.
th_eat started! ID = 4.
th_eat started! ID = 0.
Stick:  - 1  -  -  -
Stick:  - 1  2  -  -
Stick:  - 1  2  3  -
Stick:  - 1  2  3  4
Stick:  0  1  2  3  4

```

然后所有线程进入睡眠状态不再执行。从输出结果中可以发现，这是由于每个哲学家都拿起了左手边的筷子，导致的后果是谁都不可能拿到右手边的筷子。这时除了主线程是在等待子线程结束

外，其他线程都在等待某个信号灯。这种多个线程（或进程）相互竞争资源而导致不能继续执行的情形称为死锁。需要说明的是，我们在代码中拿起两根筷子的操作间隔中人为地加入睡眠操作只是大大增加了死锁发生的概率，即使没有这个睡眠，发生死锁的概率也是存在的。

12.3.1.2 简单的生产者/消费者模型

多线程并发应用程序有一个经典的模型，即生产者/消费者模型。系统中产生消息的是生产者，而处理消息的是消费者，消费者与生产者通过一个缓冲区进行消息传递。生产者产生消息后提交到缓冲区，然后通知消费者可以从中取出消息进行处理。消费者处理完信息后，通知生产者可以继续提供消息。要实现这个模型，关键在于对消费者与生产者这两个线程进行同步。也就是说，只有缓冲区中有消息时，消费者才能够提取消息，否则线程阻塞以等待消息的到来；同样，只有消息已被处理，生产者才能产生消息提交到缓冲区，否则线程阻塞以等待消息被处理。

下面给出一个实现生产者/消费者模型的例程，它使用了 POSIX 信号灯。例程被分成了三个模块，pv 模块用于实现 P/V 操作原语，pc 模块用于实现生产者工作函数、消费者工作函数及两者之间的缓冲区，main 模块则包含 main 函数，是程序的入口。

pv 模块的源代码如下：

```
/* 文件名: pv.c */
/* 说明: 简单的生产者/消费者模型 */

#include <stdio.h>
#include <semaphore.h>

void P(sem_t *sem)
{
    if(sem_wait(sem))
        perror("P operating error");
}

void V(sem_t *sem)
{
    if(sem_post(sem))
        perror("V operating error");
}
```

实际上，上述代码就是对信号灯操作函数的一个简单封装。

同时提供一个头文件以供其他模块使用：

```
/* 文件名: pv.h */
/* 说明: 简单的生产者/消费者模型 */

#ifndef PV_INCLUDED
#define PV_INCLUDED

#include <semaphore.h>

void P(sem_t *sem);
```



```
void V(sem_t *sem);

#endif
```

pc 模块的源代码如下:

```
/* 文件名: pc.c */
/* 说明: 简单的生产者/消费者模型 */

#include <stdio.h>
#include <string.h>

#include "pc.h"

/* 初始化 shared 型数据 */
void init_shared(shared *buf)
{
    sem_init(&buf->full, 0, 0); /* 满信号灯初始化为 0 */
    sem_init(&buf->empty, 0, 1); /* 空信号灯初始化为 1 */
}

/* 生产者线程 */
void *producer(void *arg)
{
    shared *share = arg;
    char area[32];
    for (;;) {
        /* 让用户输入信息 */
        printf("Input message:\n");
        fgets(area, 32, stdin);
        printf("produced item is %s\n", area);
        /* 将信息放入缓冲区 */
        P(&(share->empty));
        memcpy(share->buf, area, 32);
        V(&share->full);
    }
    return NULL;
}

/* 消费者线程 */
void *consumer(void *arg)
{
    shared *share = arg;
    char cbuf[32];
    for (;;) {
        /* 从缓冲区取出信息 */
        P(&share->full);
        memcpy(cbuf, share->buf, 32);
        V(&share->empty);
        /* 显示信息 */
        printf("consume item is %s\n", share->buf);
    }
}
```



```

    }
    return NULL;
}

```

在上述代码中,生产者在产生消息前必须等待缓冲区空,而消费者在取消息前必须等待缓冲区满。

main 模块的代码如下:

```

/* 文件名: main.c */
/* 说明: 简单的生产者/消费者模型 */

#include <stdio.h>
#include <pthread.h>

#include "pc.h"

static shared buf;

int main(void)
{
    pthread_t tid_producer;
    pthread_t tid_consumer;
    /* 初始化 shared 类型 */
    init_shared(&buf);
    /* 创建生产者和消费者线程 */
    pthread_create(&tid_producer, NULL, producer, (void *)&buf);
    pthread_create(&tid_consumer, NULL, consumer, (void *)&buf);
    /* 等待线程结束 */
    pthread_join(tid_producer, NULL);
    pthread_join(tid_consumer, NULL);
    return 0;
}

```

这个例程可以简单地用下面的命令编译:

```
gcc pv.c pc.c main.c -Wall -o pc -lrt
```

12.3.2 使用互斥体

互斥体从概念上来说类似于一个二值信号灯,即初始值为 1 的信号灯。互斥体被获取之后就不能再被获取,因此对互斥体的获取和释放操作常常称为加锁与解锁操作。

但实际上,POSIX 互斥体与信号灯具有一些不同的特性,比如互斥体有拥有者的概念,也就是说,互斥体只能由获取它的线程进行释放操作,如果违反这一原则,则结果是未定义的。互斥体在实现上可能比二值信号灯更有效率,因此在能够用互斥体的地方要尽量用互斥体。

互斥体用一个 `pthread_mutex_t` 型的变量表示。使用互斥体前要对它进行初始化,其接口函数原型如下:

```

int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);

```



其中 `mutex` 参数指向要初始化的互斥体, `attr` 参数则指向一个描述互斥体属性的结构体。函数的返回值为 0 时表示操作成功, 否则表示操作失败。`attr` 参数可以为 `NULL`, 表示使用默认属性。在线程中使用互斥体时, 一般都会将其设为默认属性。

如果互斥体是静态定义的, 则可以直接用如下方式初始化:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

这样就不需要再调用 `pthread_mutex_init` 函数进行初始化了。

互斥体不用之后应该用下面的函数进行销毁:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

其中参数 `mutex` 指向要销毁的互斥体。这个函数的返回值为 0 时表示操作成功, 否则表示操作失败。

对互斥体的主要操作函数原型如下:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

`pthread_mutex_lock` 函数用于对 `mutex` 参数指向的互斥体进行加锁。如果这时互斥体已经被锁, 则调用这个函数的线程将阻塞直到互斥体成为未锁状态。到函数返回时这个互斥体已变为已锁状态, 同时函数的调用者成为这个互斥体的拥有者。这个函数的返回值为 0 时表示操作成功, 否则表示操作失败。

`pthread_mutex_trylock` 函数也用于对 `mutex` 参数指向的互斥体进行加锁。如果这时互斥体已经被锁, 则函数以错误状态返回, `errno` 变量的值被设为 `EAGAIN`; 否则就将互斥体加锁后返回, 同时函数的调用者成为这个互斥体的拥有者。这个函数的返回值为 0 时表示操作成功, 否则表示操作失败。

`pthread_mutex_unlock` 函数则用于对 `mutex` 参数指向的互斥体进行解锁。如果这时互斥体是未锁状态或者不是当前线程所拥有的, 则结果未定义。因此互斥体必须在同一线程上成对出现。

下面用一个例程来说明线程中互斥体的用法, 代码如下:

```
/* 文件名: mutex.c */
/* 说明: 互斥体使用例程 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

#define MUTEX

#define BUF_SIZE 32 /* 缓冲区大小 */

static char buf[BUF_SIZE]; /* 缓冲区 */
```



```
#ifndef NO_MUTEX
static pthread_mutex_t mlock = PTHREAD_MUTEX_INITIALIZER; /* 互斥体变量 */
#endif

/* 进行随机长度的延时以人为增加程序的执行时间 */
/* 编译时不可以进行优化, 否则无作用 */
static void delay()
{
    int i;
    int t = rand() >> 8;
    for (i = 0; i < t; i++) i = i;
}

/* 设置缓冲区中的字符 */
void set_buf(char ch)
{
    char *p;
#ifdef NO_MUTEX
    pthread_mutex_lock(&mlock);
#endif
    for (p = buf; p < buf+BUF_SIZE; p++) {
        *p = ch;
        delay(); /* 延时 */
    }
#ifdef NO_MUTEX
    pthread_mutex_unlock(&mlock);
#endif
}

/* 检查缓冲区的内容 */
void check(void)
{
    char ch;
    char *p;
#ifdef NO_MUTEX
    pthread_mutex_lock(&mlock);
#endif
    ch = buf[0]; /* 保存第一个字符 */
    for (p = buf+1; p < buf+BUF_SIZE; p++) {
        if (*p != ch) { /* 如果与第一个不一致 */
            /* 输出错误 */
            for (p = buf; p < buf+BUF_SIZE; p++) putchar(*p);
            putchar('\n');
            break;
        }
    }
#ifdef NO_MUTEX
    pthread_mutex_unlock(&mlock);
#endif
}
```



```

/* 线程工作函数 */
void *thread_fun(void *arg)
{
    int ch = (int)arg;
    for (;;) set_buf((char)ch);
}

int main()
{
    int res;
    int ch;
    pthread_t tid;
    memset(buf, 0, BUF_SIZE);
    for (ch = 'A'; ch <= 'F'; ch++) {
        res = pthread_create(&tid, NULL, thread_fun, (void *)ch);
        if (res != 0) {
            perror("Thread creation failed");
            exit(EXIT_FAILURE);
        }
    }
    for (;;) {
        check();
        sleep(1);
    }
    return 0;
}

```

在上述例程中定义了一块全局内存缓冲区 `buf`, 然后创建了若干个线程分别向缓冲区写入不同的数据, 主线程则每隔一段时间去检查一次缓冲区中的内容。每个线程在写入时总是试图将整个缓冲区写为相同的字符, 检查缓冲区时如果发现字符不一致, 就认为有错误发生并输出信息。

在没有使用互斥体的情况下, 也就是在程序中定义了 `NO_MUTEX` 的情况下, 线程对缓冲区的写入过程可能被其他线程打断, 因此当检查缓冲区时有可能发现字符不一致的情况, 从而输出错误信息。如果使用了互斥体, 则每个线程在写入前都会试图获取互斥体, 写完全部字符后才释放互斥体, 这样就可以保证一个线程的写入过程不被其他线程打断, 从而不会发生缓冲区中字符不一致的情况。

这种需要对共享数据进行独占式访问的代码通常称为临界区, 互斥体的一个主要用途就是保护临界区。

在例程中, 为了让线程的写入过程被打断的概率增加, 我们人为地让它每写一个字符就进行一个随机长度的延时。由于计算机的运行速度很快, 如果没有人为的延时, 则对缓冲区的写入将在很短的时间内完成, 被其他线程打断的概率非常小, 这也是实际应用中常见的情况。互斥体在实现上考虑到了这一点, 获取互斥体时, 如果能够获取成功, 则执行的代码量很少, 对运行效率的影响也很小; 如果不能获取, 则执行的代码量大大增加, 并且会引起线程状态变化等复杂操作。

12.3.3 使用条件变量

前面说明了互斥体对全局数据访问的保护, 但实际应用中, 还有一些情况, 比如线程正在等待共享数据内某个条件出现, 这时必须先解锁, 否则其他线程不可能更改共享数据。实现时可以循环

检测共享数据，但在检测前要加锁，检测后又要解锁。可以想象，这样的代码效率多么低。

因此，在这种情况下，我们需要有一种方法，使得当线程在等待满足某些条件时进入睡眠状态，一旦条件满足，线程就应该被唤醒继续执行。这个方法就是使用 POSIX 条件变量。

POSIX 条件变量由一个 `pthread_cond_t` 型变量表示。使用条件变量前要对它进行初始化，所用的接口函数原型如下：

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

其中 `cond` 参数指向要初始化的条件变量，`attr` 参数则指向描述条件变量属性的结构体。函数返回 0 表示操作成功，非 0 表示操作失败。`attr` 参数可以为 `NULL`，表示使用默认属性。在线程中使用条件变量时一般都用默认属性。

如果条件变量是静态定义的，则可以直接用如下方式初始化：

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

这样就不需要再调用 `pthread_cond_init` 函数进行初始化了。

条件变量不用之后应该用下面的函数进行销毁：

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

其中参数 `cond` 指向要销毁的条件变量。这个函数的返回值为 0 时表示操作成功，否则表示操作失败。

当程序中需要等待一个条件变量时，可以用下面的函数：

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,  
    const struct timespec *abstime);  
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

其中参数 `cond` 指向要等待的条件变量。注意这些函数都有一个指向互斥体的参数 `mutex`，说明条件变量必须与互斥体搭配使用。调用这些函数时，首先指定的互斥体将被释放，然后线程将阻塞，等待条件变量的触发。函数返回前，互斥体重新被线程获取。这些函数返回 0 表示操作成功，非 0 表示操作失败。

`pthread_cond_timedwait` 函数与 `pthread_cond_wait` 函数的区别在于，前者有一个由参数 `abstime` 指定的超时时间限制，当线程阻塞的时间超过了这个时间就会自动醒来，函数以错误状态返回，`errno` 变量的值被设为 `ETIMEDOUT` 以表示发生了超时。

触发一个条件变量可用以下函数：

```
int pthread_cond_signal(pthread_cond_t *cond);  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

`pthread_cond_signal` 可以唤醒一个或多个正在等待 `cond` 参数所指向的条件变量的线程，而 `pthread_cond_broadcast` 则可以唤醒全部正在等待 `cond` 参数所指向的条件变量的线程。这些函数返回 0 表示操作成功，非 0 表示操作失败。

下面给出一个例程来说明条件变量及互斥体的配合使用，代码如下：

```
/* 文件名: cond.c */
/* 说明: 条件变量使用例程 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER; /* 互斥体 */
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER; /* 条件变量 */

static int x = 0;

/* 消费者线程工作函数 */
static void *thread_eat(void *arg)
{
    int id = (int)arg;
    printf("thread %d run\n", id);
    while (1) {
        pthread_mutex_lock(&lock);
        /* 判断条件是否满足, 如果 x <= 0, 则等待条件变量 cond */
        while (x <= 0) {
            printf("thread %d will wait on cond\n", id);
            pthread_cond_wait(&cond, &lock);
            printf("thread %d wakeup\n", id);
        }
        x--;
        printf("thread %d eat one, now x = %d\n", id, x);
        pthread_mutex_unlock(&lock);
        sleep(1);
    }
    return NULL;
}

/* 生产者线程工作函数 */
static void *thread_feed(void *arg)
{
    while (1) {
        pthread_mutex_lock(&lock);
        x += 4;
        /* 触发条件变量 */
        pthread_cond_broadcast(&cond);
        printf("thread feed set x = %d\n", x);
        pthread_mutex_unlock(&lock);
        sleep(2);
    }
    return NULL;
}

int main()
{

```



```

int i;
pthread_t tid;
for (i = 0; i < 3; i++) {
    if (pthread_create(&tid, NULL, thread_eat, (void *)i)) {
        perror("pthread_create");
        return -1;
    }
}
sleep(1); /* 保证消费者线程先运行 */
if (pthread_create(&tid, NULL, thread_feed, NULL)) {
    perror("pthread_create");
    return -1;
}
pthread_join(tid, NULL);
return 0;
}

```

在这个例程中有一个生产者线程 `thread_feed` 和多个消费者线程 `thread_eat`。生产者每隔一段时间将产生 4 个资源（用全局变量 `x` 的值加 4 来代表），消费者则每隔一段时间消耗 1 个资源（用全局变量 `x` 的值减 1 来代表）。只有当资源个数大于 0 时消费者才能进行消耗，否则就要等待生产者投放资源。

例程的一个运行结果如下：

```

thread 1 run
thread 0 run
thread 0 will wait on cond
thread 1 will wait on cond
thread 2 run
thread 2 will wait on cond
thread feed set x = 4
thread 1 wakeup
thread 1 eat one, now x = 3
thread 2 wakeup
thread 2 eat one, now x = 2
thread 0 wakeup
thread 0 eat one, now x = 1
thread 1 eat one, now x = 0
thread 0 will wait on cond
thread 2 will wait on cond

```

从中可以很明显地看到，当条件变量触发时多个线程都被唤醒。如果将生产者工作函数中触发条件变量的函数改为 `pthread_cond_signal`，则结果变为如下：

```

thread 0 run
thread 0 will wait on cond
thread 1 run
thread 1 will wait on cond
thread 2 run
thread 2 will wait on cond
thread feed set x = 4

```




```

thread 0 wakeup
thread 0 eat one, now x = 3
thread 0 eat one, now x = 2
thread feed set x = 6
thread 1 wakeup
thread 1 eat one, now x = 5
thread 0 eat one, now x = 4
thread 0 eat one, now x = 3
thread 1 eat one, now x = 2

```

这时我们发现每次投放资源时，只有一个睡眠中的线程被唤醒。需要指出的是，POSIX 标准只规定 `pthread_cond_signal` 函数唤醒至少一个睡眠中的线程，并没有规定只唤醒一个，因此编程时最好不要利用这个特性。

由于我们的例程是无限循环的，以上的运行结果都只摘取了前面的若干行。

从例程中我们也可以看到，条件变量是与互斥体一起配合使用的。

12.4 多线程并发程序设计

在操作系统发展的早期，没有线程的概念，因此只能用多进程的方式实现并发程序设计。有了线程之后，就可以考虑使用线程进行并发程序设计，以得到更高的执行效率。当然，多进程与多线程各有其优缺点，需要根据实际情况考虑具体的设计，也可能会将多进程与多线程结合起来使用。

本节将在前面几节的基础上，通过例程进一步说明多线程的几个典型应用。

12.4.1 多线程并发服务器应用

在 `socket` 编程一章，我们已经给出了多进程并发的服务器例程，也给出了使用多路复用的单进程服务器例程，下面将其改造为使用多线程对多个客户端同时响应的方式。

例程代码如下：

```

/* 文件名: tcp_thread.c */
/* 说明: TCP 多线程服务器例程 */

#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <pthread.h>

#define DEFAULT_PORT 8888 /* 默认端口 */
#define BACKLOG 5 /* 连接请求队列长度 */
#define BUF_SIZE 128 /* 缓冲区大小 */

```



```
/* 用于保存一个客户端的信息 */
struct client {
    int sockfd; /* socket 描述符 */
    char buf[BUF_SIZE+1]; /* 缓冲区 */
    char ip[16]; /* IP 地址 (字符串) */
    unsigned short port; /* 端口号 */
};

static int send_str(int sock, const char *str);
static int recv_str(int sock, char *buf, int max_len);
static int send_byte(int sock, const char *buf, int len);
static int recv_byte(int sock, char *buf, int len);
static int eat_byte(int sock, int len);

static int start_server(int port);
static int start_client(const char *addr, int port);

static void *thread_work(void *arg);

static char g_buf[BUF_SIZE];

/* 主函数 */
int main(int argc, char *argv[])
{
    int i;
    int r;
    int port;
    int server_mode;
    char *addr;
    port = DEFAULT_PORT;
    server_mode = 0;
    addr = "127.0.0.1";
    /* 对命令行参数进行分析 */
    for (i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-s") == 0) { /* 如果是 -s */
            server_mode = 1; /* 说明要以服务器方式启动 */
        } else if (strcmp(argv[i], "-p") == 0) { /* 如果是 -p */
            i++; /* 下一个参数 */
            if (i >= argc) {
                printf("Error: must specify port number!\n");
                return -1;
            }
            port = atoi(argv[i]); /* 是端口号 */
            if (port == 0) {
                printf("Error: wrong port number!\n");
                return -1;
            }
        } else { /* 否则 */
            addr = argv[i]; /* 是 IP 地址 */
        }
    }
}
```



```

    }
    if (server_mode) {
        r = start_server(port); /* 启动服务器 */
    } else {
        r = start_client(addr, port); /* 启动客户端 */
    }
    return r;
}

/* 服务器程序 */
int start_server(int port)
{
    int r;
    int val;
    int sock_listen, sock;
    struct sockaddr_in host_addr, peer_addr;
    pthread_t tid;
    socklen_t socklen;
    struct client *client;
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    /* 打开 socket */
    sock_listen = socket(PF_INET, SOCK_STREAM, 0);
    if (sock_listen < 0) {
        perror("start_server: socket()");
        return -1;
    }
    /* 设置地址可重用 */
    val = 1;
    setsockopt(sock_listen, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
    /* 绑定 */
    memset(&host_addr, 0, sizeof(host_addr));
    host_addr.sin_family = AF_INET;
    host_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    host_addr.sin_port = htons(port);
    r = bind(sock_listen, (struct sockaddr *)&host_addr, sizeof(host_addr));
    if (r < 0) {
        perror("start_server: bind()");
        close(sock_listen);
        return -1;
    }
    /* 侦听 */
    r = listen(sock_listen, BACKLOG);
    if (r < 0) {
        perror("start_server: listen()");
        close(sock_listen);
        return -1;
    }
    printf("Server start listening on port %d.\n", port);
    /* 主循环 */

```

```

    for ( ; ; ) {
        /* 接受 */
        socklen = sizeof(peer_addr);
        sock = accept(sock_listen, (struct sockaddr *)&peer_addr, &socklen);
        if (sock < 0) {
            perror("start_server: accept()");
            continue;
        }
        /* 接受成功则生成新线程 */
        client = malloc(sizeof(struct client));
        client->sockfd = sock;
        /* 保存对端 IP 地址和端口以备将来显示 */
        strcpy(client->ip, inet_ntoa(peer_addr.sin_addr));
        client->port = ntohs(peer_addr.sin_port);
        if (pthread_create(&tid, &attr, thread_work, (void *)client)) {
            perror("pthread_create");
            close(sock);
        }
    }
    pthread_attr_destroy(&attr);
    return 0;
}

/* 启动客户端 */
int start_client(const char *addr, int port)
{
    int r;
    int sock;
    struct sockaddr_in server_addr;
    /* 打开 socket */
    sock = socket(PF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("start_client: socket()");
        return -1;
    }
    /* 连接 */
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    if (!inet_aton(addr, &server_addr.sin_addr)) {
        perror("start_client: inet_aton()");
        close(sock);
        return -1;
    }
    server_addr.sin_port = htons(port);
    r = connect(sock, (struct sockaddr *)&server_addr, sizeof(server_addr));
    if (r < 0) {
        perror("start_client: connect()");
        close(sock);
        return -1;
    }
    /* 接收与发送循环 */

```

```

    for ( ; ; ) {
        /* 接收一个字符串 */
        if (recv_str(sock, g_buf, sizeof(g_buf)) <= 0) break;
        /* 显示接收到的字符串 */
        printf("> %s\n", g_buf);
        /* 让用户输入一行数据 */
        fgets(g_buf, sizeof(g_buf), stdin);
        g_buf[strlen(g_buf)-1] = '\0';
        /* 将数据发送给服务器 */
        if (send_str(sock, g_buf) <= 0) break;
    }
    close(sock);
    return 0;
}

/* 工作线程，用于与一个客户端交换数据 */
void *thread_work(void *arg)
{
    struct client *client = (struct client *)arg;
    int sock = client->sockfd;
    char *buf = client->buf;
    printf("Accept remote %s:%d\n", client->ip, client->port);
    /* 收发数据循环 */
    for ( ; ; ) {
        /* 发送字符串 OK 并接收来自客户端的数据 */
        if (send_str(sock, "OK") <= 0
            || recv_str(sock, buf, BUF_SIZE) <= 0) {
            printf("%s:%d error or disconnected, "
                "socket is closing...\n",
                client->ip, client->port);
            close(sock);
            break;
        }
        /* 输出来自客户端的数据 */
        printf("%s:%d> %s\n", client->ip, client->port, buf);
    }
    /* 退出时释放内存 */
    free(client);
    return NULL;
}

/* 发送字符串 */
int send_str(int sock, const char *str)
{
    int byte;
    int len;
    int len_send;
    /* 首先发送一个整数，表示字符串的长度 */
    len = strlen(str);
    len_send = htonl(len);
    byte = send_byte(sock, (char *)&len_send, sizeof(len_send));

```



```
    if (byte < 0) return -1;
    /* 然后发送字符串本身 */
    byte = send_byte(sock, str, len);
    if (byte < 0) return -1;
    return byte;
}

/* 接收字符串 */
int recv_str(int sock, char *buf, int max_len)
{
    int byte;
    int len;
    int len_recv;
    int len_diff;
    len_diff = 0;
    /* 首先接收一个整数, 这个整数表示字符串长度 */
    byte = recv_byte(sock, (char *)&len_recv, sizeof(len_recv));
    if (byte < 0) return -1;
    len = ntohs(len_recv);
    if (len > max_len) {
        len_diff = len-max_len;
        len = max_len;
    }
    /* 然后接收字符串 */
    byte = recv_byte(sock, buf, len);
    if (byte <= 0) return -1;
    buf[byte] = '\0';
    /* 如果实际接收的字符数小于字符串的长度, 则消耗掉多余的字符 */
    if (len_diff > 0) eat_byte(sock, len_diff);
    return byte;
}

/* 可靠发送 len 个字节 */
int send_byte(int sock, const char *buf, int len)
{
    int rc;
    int byte;
    for (byte = 0; byte < len; byte += rc) {
        rc = send(sock, buf+byte, len-byte, MSG_NOSIGNAL);
        if (rc < 0 && errno != EINTR) {
            byte = -1;
            break;
        }
    }
    return byte;
}

/* 可靠接收 len 个字节 */
int recv_byte(int sock, char *buf, int len)
{
    int rc;
```



```

    int byte;
    for (byte = 0; byte < len; byte += rc) {
        rc = recv(sock, buf+byte, len-byte, MSG_NOSIGNAL);
        if (rc == 0) break;
        if (rc < 0 && errno != EINTR) {
            byte = -1;
            break;
        }
    }
    return byte;
}

/* 消耗 len 个字节 */
int eat_byte(int sock, int len)
{
    int rc;
    int byte;
    char buf[32];
    for (byte = 0; len-byte > 32; byte += rc) {
        rc = recv(sock, buf, 32, MSG_NOSIGNAL);
        if (rc == 0) break;
        if (rc < 0 && errno != EINTR) return -1;
    }
    if (rc != 0) byte += recv_byte(sock, buf, len-byte);
    return byte;
}

```

由于多个线程共享全局变量,所以在上述例程中不能再像多进程那样用一个全局的缓冲区来处理多个客户端的数据收发,这是其设计要点之一。其次,接受客户端以后得到的新 `socket` 虽然不在主线程中使用,但不能关闭,因为创建新线程并不增加对已打开文件的引用计数。

12.4.2 消费者/生产者模型

在上一节中提出的消费者/生产者模型比较简单,缓冲区中只能容纳一条消息。生产者每提交一条消息到缓冲区中,就会通知消费者,等消费者取走消息之后才能提交下一条消息。同样,消费者也必须等待生产者提交一条消息后才能进行处理。这种设计的效率是比较低下的。

如果将缓冲区设计为一个先进先出的队列,可以同时容纳多条消息,那么只要缓冲区不满,生产者就可以提交消息;同时,只要缓冲区不空,消费者就可以取出消息进行处理。这将大大提高整个流程的效率。

在实现上,我们可以利用信号灯有计数的特性,用信号灯的值表示缓冲区中消息的个数及空闲空间的个数。但这时由于生产者和消费者可能同时访问缓冲区,故需要再用一个互斥体来进行保护。因此对一个缓冲区需要定义以下三个同步变量:

```

sem_t full; /* 表示缓冲区中消息的个数 */
sem_t empty; /* 表示缓冲区中的空闲空间 (还能容纳的消息个数) */
pthread_mutex_t lock; /* 同步对缓冲区的访问 */

```

这些同步变量应进行如下的初始化:



```
sem_init(&buf->full, 0, 0); /* 缓冲区中消息数为 0 */
sem_init(&buf->empty, 0, N); /* 缓冲区中的空闲空间数为 N, 即它的容量 */
pthread_mutex_init(&lock, NULL); /* 初始化互斥体 */
```

改进后生产者和消费者的代码原型（不是完整代码）如下：

```
/* 生产者 */
void *producer(void *arg)
{
    item *item; /* 消息 */
    for (;;) {
        item = produce_item(); /* 生产消息 */
        sem_wait(&empty); /* 获得表示空闲空间的信号灯 */
        pthread_mutex_lock(&lock); /* 加锁 */
        insert_item(item); /* 将消息放入缓冲区 */
        pthread_mutex_unlock(&lock); /* 解锁 */
        sem_post(&full); /* 释放表示消息个数的信号灯 */
    }
    return NULL;
}

/* 消费者 */
void *consumer(void *arg)
{
    item *item; /* 消息 */
    for (;;) {
        sem_wait(&full); /* 获得表示消息个数的信号灯 */
        pthread_mutex_lock(&lock); /* 加锁 */
        item = remove_item(); /* 取得消息 */
        pthread_mutex_unlock(&lock); /* 解锁 */
        sem_post(&empty); /* 释放表示空闲空间的信号灯 */
        consume_item(item); /* 处理消息 */
    }
    return NULL;
}
```

12.4.3 线程池应用

在上述生产者/消费者的多线程处理模型中，实际上消费者线程可能不只一个。例如在前述多线程的服务器应用中，每接受一个新的客户端，我们就创建一个新的线程去处理。这样做的问题在于，尽管创建线程比创建进程减少很多开销，但毕竟还是一个耗时的的工作，将在一定程度上影响效率。

在很多解决方案中，应用程序初始化时就会创建一批线程，然后每个线程都从任务队列中提取任务执行，这种方案称为线程池。线程池应用的实现依赖于很多关键设计，如任务队列的设计、任务的数据表示等。

下面我们给出一个例程来说明线程池的应用，代码如下：

```
/* 文件名: thread_pool.c */
/* 说明: 线程池应用 */
```




```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

#define NAME_SIZE 10 /* 任务名长度 */

/* 任务队列节点，任务队列用链表实现 */
typedef struct task {
    void (*handler)(void *); /* 任务处理函数 */
    void *arg; /* 传给处理函数的参数 */
    struct task *next; /* 指向下一个节点 */
} task_t;

/* 线程信息 */
typedef struct thread_info {
    pthread_t id; /* 线程句柄 */
    int index; /* 线程编号 */
} thread_info_t;

/* 线程池数据 */
typedef struct pool_struct {
    int num_threads; /* 线程池的大小 */
    int max_task; /* 任务队列大小 */
    sem_t empty;
    sem_t full;
    pthread_mutex_t lock;
    thread_info_t *threads; /* 线程池中的线程 */
    task_t *head; /* 队列头 */
    task_t *tail; /* 队列尾 */
} pool_struct_t;

static int pool_add_task(pool_struct_t *pool, task_t *task);
static void *process_task(void *arg);
static void task_handler(void *arg);
static void task_exit(void *arg);

/* 创建任务并初始化 */
static task_t *create_task(void (*routine)(void *), char name[])
{
    task_t *task;
    /* 分配空间以存储任务数据 */
    task = malloc(sizeof(task_t));
    if (!task) return NULL;
    /* 分配空间以存储任务名 */
    task->arg = malloc(NAME_SIZE);
    if (!task->arg) {
        free(task);
    }
}

```



```
        return NULL;
    }
    task->handler = routine;
    strncpy(task->arg, name, NAME_SIZE);
    task->next = NULL;
    return task;
}

/* 销毁任务 */
static void destroy_task(task_t *task)
{
    /* 释放空间 */
    free(task->arg);
    free(task);
}

/* 创建线程池并初始化 */
static pool_struct_t *create_pool(int num_threads, int max_task)
{
    int i;
    pool_struct_t *pool;
    printf("init thread pool ...\n");
    /* 分配线程池数据的空间 */
    pool = (struct pool_struct *)malloc(sizeof(struct pool_struct));
    if (pool == NULL) {
        printf("Unable to malloc thread pool!\n");
        return NULL;
    }
    /* 分配空间以存放线程信息 */
    pool->threads = (thread_info_t *)malloc(sizeof(thread_info_t)*num_threads);
    if (pool->threads == NULL) {
        printf("Unable to malloc thread info array\n");
        free(pool);
        return NULL;
    }
    pool->num_threads = num_threads;
    pool->max_task = max_task;
    pool->head = pool->tail = NULL;
    sem_init(&pool->empty, 0, max_task);
    sem_init(&pool->full, 0, 0);
    pthread_mutex_init(&pool->lock, NULL);
    /* 创建线程 */
    for (i = 0; i < num_threads; i++) {
        pool->threads[i].index = i;
        if (pthread_create(&(pool->threads[i].id),
            NULL, process_task, (void*)pool)) {
            perror("pthread_create");
            free(pool->threads);
            free(pool);
            return NULL;
        }
    }
}
```



```

    }
    return pool;
}

/* 销毁线程池 */
static void destroy_pool(pool_struct_t *pool)
{
    int i;
    task_t *task;
    /* 清空任务队列中的任务 */
    pthread_mutex_lock(&pool->lock); /* 加锁 */
    do {
        task = pool->head;
        pool->head = task->next;
        destroy_task(task);
    } while (!pool->head);
    pthread_mutex_unlock(&pool->lock); /* 解锁 */
    /* 向任务队列开头加入退出任务以使工作线程退出 */
    for (i = 0; i < pool->num_threads; i++) {
        task = create_task(task_exit, "EXIT");
        pool_add_task(pool, task);
    }
    for (i = 0; i < pool->num_threads; i++) {
        pthread_join(pool->threads[i].id, NULL);
    }
    /* 销毁信号灯和互斥体 */
    sem_destroy(&pool->empty);
    sem_destroy(&pool->full);
    pthread_mutex_destroy(&pool->lock);
    /* 释放内存 */
    free(pool->threads);
    free(pool);
}

/* 向任务队列添加任务 */
int pool_add_task(pool_struct_t *pool, task_t *task)
{
    /* 获取表示队列空闲空间的信号灯 */
    sem_wait(&pool->empty);
    /* 添加到链表 */
    pthread_mutex_lock(&pool->lock); /* 加锁 */
    if (pool->head == NULL) { /* 第一个节点 */
        pool->tail = pool->head = task;
    } else {
        pool->tail->next = task;
        pool->tail = task;
    }
    pthread_mutex_unlock(&pool->lock); /* 解锁 */
    /* 释放表示队列中任务数的信号灯 */
    sem_post(&pool->full);
    return 0;
}

```



```
}

/* 线程工作函数，处理任务 */
void *process_task(void *arg)
{
    int i, index = -1;
    task_t *task;
    pthread_t myid;
    pool_struct_t *pool = (struct pool_struct *)arg;
    /* 查找线程信息数组以得到自己的编号 */
    myid = pthread_self();
    for (i = 0; i < pool->num_threads; i++) {
        if (myid == pool->threads[i].id) {
            index = pool->threads[i].index;
            break;
        }
    }
    printf("thread %d ready\n", index);
    /* 取任务并执行 */
    for (;;) {
        /* 获取表示队列中任务数的信号灯 */
        sem_wait(&pool->full);
        pthread_mutex_lock(&pool->lock); /* 加锁 */
        /* 从链表中取出任务 */
        task = pool->head;
        pool->head = task->next;
        pthread_mutex_unlock(&pool->lock); /* 解锁 */
        /* 释放表示队列空闲空间的信号灯 */
        sem_post(&pool->empty);
        /* 执行任务 */
        printf("thread %d take task %s ...\n", index, (char *)task->arg);
        (*(task->handler))(task->arg);
        /* 销毁任务 */
        destroy_task(task);
    }
    return NULL;
}

/* 任务处理函数 */
void task_handler(void *arg)
{
    char *ptr = (char *)arg;
    sleep(1);
    printf("task %s DONE!\n", ptr);
}

/* 特殊任务：退出线程 */
void task_exit(void *arg)
{
    char *ptr = (char *)arg;
    printf("task %s DONE!\n", ptr);
}
```



```

    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    int i;
    char name[NAME_SIZE];
    task_t *task;
    pool_struct_t *pool;
    /* 创建线程池 */
    pool = create_pool(3, 5);
    /* 向任务队列增加任务, 这里主线程就是生产者角色 */
    for (i = 0; i < 10; i++) {
        sprintf(name, "TASK %d", i);
        task = create_task(task_handler, name);
        if (!task) {
            printf("Unable to create task\n");
            destroy_pool(pool);
            return -1;
        }
        pool_add_task(pool, task);
    }
    destroy_pool(pool);
    return 0;
}

```

在这个例程中, 任务由一个任务处理函数和传给它的参数组成, 线程取得这个任务后, 只需要执行这个处理函数即可完成任务。通过这种方式, 线程可以与具体的任务内容脱离, 也就是说, 不管哪个线程都可以执行队列中的任何任务, 一个任务最终由哪个线程取得完全取决于系统的调度。

任务列表在例程中使用的是单链表数据结构, 实际上, 由于最大任务个数是固定的, 因此也可以用数组实现。

主线程充当了生产者的角色, 负责生产任务, 而线程池中的线程则充当了消费者的角色。

当线程池销毁时, 我们采用了一种特殊的做法, 即向任务队列中放入退出任务, 线程执行这个任务的结果就是退出。采用这种办法就可以简单地实现销毁线程池操作与各个工作线程的同步。

程序的一个运行结果如下:

```

init thread pool ...
thread 1 ready
thread 0 ready
thread 1 take task TASK 0 ...
thread 0 take task TASK 1 ...
thread 2 ready
thread 2 take task TASK 2 ...
task TASK 0 DONE!
thread 1 take task TASK 3 ...
task TASK 2 DONE!
thread 2 take task TASK 4 ...
task TASK 1 DONE!

```



```
thread 0 take task TASK 5 ...
task TASK 3 DONE!
thread 1 take task TASK 7 ...
task TASK 5 DONE!
thread 0 take task TASK 8 ...
task TASK 4 DONE!
thread 2 take task TASK 9 ...
task TASK 7 DONE!
thread 1 take task EXIT ...
task EXIT DONE!
task TASK 9 DONE!
thread 2 take task EXIT ...
task EXIT DONE!
task TASK 8 DONE!
thread 0 take task EXIT ...
task EXIT DONE!
```

从结果中可以清楚地看到各个线程取得任务并执行任务以及最后退出的过程。



Part

第 4 部分 内核与驱动编程

第 13 章 内核编程初步

第 14 章 内核编程接口

第 15 章 Linux 2.6 设备模型

第 16 章 Linux 驱动实例详解



第 13 章 内核编程初步

从应用程序的编程到内核驱动的编程，中间有很多观念和方法的转变。本章的目的就是让有一定应用编程经验的读者完成这一转变，掌握 Linux 内核编程的基本方法，以便进一步了解内核的工作原理和编程接口，为内核态驱动的编写打下基础。

本章的内容可分为三个主要部分：首先，介绍的是一些基本概念、内核编程的特点及与应用编程的对比；然后，介绍内核模块编程的方法，这是内核编程的最常用形式；最后，通过一个字符设备驱动的例子，介绍内核驱动编程的一般原理和一些基本的编程接口。

13.1 从用户态到内核态

这一节的主要内容是介绍内核编程的一些基础知识，以及内核编程与应用编程的区别，为后面各节的内容打下基础。

13.1.1 实模式与保护模式

现代的 CPU 一般都有实模式和保护模式两种访问内存的方式。在实模式下，CPU 将指令中的内存地址直接送到地址总线上，以读写相应内存单元的内容。这种直接对内存有效的地址称为物理地址。也就是说，在实模式下，访问内存的指令中可以直接使用物理地址。

保护模式下情况要复杂一些，指令中使用的地址不是内存的物理地址，而是所谓的虚拟地址。CPU 在访问内存时，要先将指令中的虚拟地址转换为物理地址，再送到地址总线上，转换的过程称为地址映射。这种映射操作一般是通过 MMU (Memory Management Unit, 内存管理单元) 进行的。

MMU 是专门用于地址映射的硬件。映射操作采用查表的方式，以虚拟地址作为索引去查找对应的物理地址，所查的表称为页表，也存放在内存里。显然，更换不同的页表，同一个虚拟地址对应的物理地址就是不同的，实际上访问的就是不同的内存单元。

CPU 刚复位之后都工作在实模式下，MMU 不做地址映射操作。通过软件的方式设置好页表，再打开 MMU 的地址映射功能，就切换到了保护模式下，这正是操作系统内核启动时工作的一部分。

保护模式给操作系统带来的好处很多。内核可以为每个进程设置不同的页表，使每个进程都在自己的虚拟地址空间中运行，互不影响。页表中还可以保存内存的访问权限，当某个进程进行非法的内存访问时，将引发 MMU 的异常，执行流程自动进入到内核的异常处理程序中，内核将当前进程结束，而不会导致整个系统的崩溃。一般情况下，内核在切换到保护模式下后，不会再切换回实模式。

13.1.2 用户态与内核态

在保护模式下，CPU 还有至少两种工作状态，可称为用户态和内核态。在用户态下，CPU 可以执行的指令是受限的，诸如设置页表、切换工作模式、修改中断状态、访问外部端口等操作都会引发异常。在内核态下，CPU 可以执行任何所支持的指令。

内核对每个进程都会设置一个独立的页表，内核自己访问内存也需要一个页表。当 CPU 处于用户态时，使用的是当前进程的页表。当 CPU 处于内核态时，使用的是内核的页表。内核还为每个进程设立两个栈：用户栈和内核栈，分别在用户态和内核态时使用。

CPU 从用户态进入内核态有以下几种方式。

一、中断

中断是 CPU 与外部设备交互的一种方式。在硬件上，外部设备与 CPU 通过中断线连接，当设备的状态发生变化，需要通知 CPU 时，在中断线上输出电信号。CPU 响应中断信号，进入中断处理过程，跳转到预先指定的位置执行，同时进入内核态。

二、异常

CPU 执行指令时发现不正常的情况，如非法指令、未知指令、非法的内存访问、被零除等，将引发异常，进入异常处理过程，跳转到预先指定的位置执行，同时进入内核态。

三、陷阱

软件上执行一些特定的指令，将使 CPU 跳转到预先指定的位置执行，同时进入内核态。

以上三种进入内核态的方式有类似的地方，即 CPU 中止当前的执行流程，跳转到预先指定的位置执行，因此有时又将其统称为中断，而把第一种称为外部中断，第二种称为内部中断，第三种称为软中断。本章中将采用中断、异常和陷阱三者分立的术语，原因是后一种说法中的软中断容易与内核的软中断机制混淆。

当中断、异常或陷阱处理流程结束后，通过软件指令可以返回到原来中止执行处继续执行，CPU 回到原来的工作状态。必须注意，在内核态时，以上三种情况也都是有可能发生的。操作系统内核的一项重要工作就是处理各种中断和异常。

应用程序的代码在用户态执行，而内核的代码在内核态执行。应用程序到内核的编程接口通常就用陷阱的方式实现，进程在用户态主动执行指令，陷入内核态，内核处理完毕后返回到用户态。这种编程接口称为系统调用，通常都被包装成 C 语言函数的形式。

进程通过系统调用进入内核态，执行内核的代码，这段代码的执行属于进程执行过程的一部分，因此称它在这个进程的上下文中执行。与此相反，中断和异常的处理过程是独立的，不属于任何进程，这个过程中执行的代码称为在中断上下文中执行。需要注意的是，同一段内核代码有可能同时在多个进程的上下文中执行。

13.1.3 内核编程的特点

内核编程与应用程序编程的特点对比如表 13.1 所示。

表 13.1 内核编程与应用编程对比

内核编程	应用编程
代码运行在内核态	代码运行在用户态
程序的错误可能导致整个系统死锁或崩溃	程序的错误可能导致进程崩溃，不会影响内核
可以直接访问硬件	访问外部设备要通过系统调用接口
代码不管在多少个进程的上下文中执行，所访问的全局变量都是同一个存储空间	代码在多个进程内执行时，每个进程各自有全局变量的存储空间，互不影响
一般没有浮点数操作	有浮点数操作
基本只能用 C 语言（和汇编语言）	可用各种语言编程
不能使用系统调用	可以调用系统调用

由于应用程序可以使用系统调用，而系统调用是有标准可循的（Linux 操作系统一般都支持 POSIX 标准的系统调用），因此应用程序可以很容易地在不同版本的操作系统之间移植，甚至移植到另外一种操作系统。而内核编程所使用的接口函数都是内核本身提供的，没有统一的标准，因此不同版本的内核之间差异较大。本章内如无特别说明，所介绍的内容针对的都是 Linux 2.6.30 版本的内核。

另外，内核中有很多代码是平台相关的，对于不同的目标平台有不同的分支。如无特殊说明，本章在节录相关内核代码时，均选择 ARM 架构分支。

内核代码中大量地利用宏来模拟函数，多数情况下两者在使用上没有区别。为了明确接口参数和返回值的类型，本章在介绍内核编程接口时尽量采用函数的形式，但实际上可能是一个宏。

由于 Linux 内核中大量使用了 C 语言的 GNU 扩展语法，因此只能用 gcc 来编译。对于 Linux 2.6 以上版本的内核，必须使用 gcc 3.4 及以上版本才有可能编译通过。

13.1.4 内核模块与驱动

在 Linux 操作系统上进行内核编程时，可以将代码编译成一个独立的模块。模块以文件的形式保存，在内核启动之后，可以动态地加载进内核，如果模块没有被使用，还可以从内核中卸载。这种代码模块称为内核模块。

内核模块在加载之后就可以视为内核的一部分，其代码运行在内核态。内核模块可以使用内核所导出的函数，也可以提供一些函数让内核调用。加载和卸载内核模块的操作必须由 root 用户进行。

因为内核模块所用的函数接口必须与内核提供的接口完全一致才能正确加载，因此一般情况下都要放在内核的编译体系内进行编译。Linux 2.6 内核上，编译生成的内核模块文件的后缀为 .ko。内核模块的源码也可以与内核编译成一个整体，这一点只需要通过修改内核的配置就可以做到，不需要改动模块的源码。

设备驱动程序是指与具体的外部设备有关、通过提供相关的接口调用、让内核或应用程序能够正确地使用设备的程序。在 Linux 上，设备驱动程序常常以内核模块的形式出现，驱动程序向内核提供必要的功能接口，让内核可以正确地管理特定设备。如果内核将底层接口暴露给了应用程序，那么也可以提供一个共享库或服务来管理连接在此接口上的设备，这种形式称为用户态设备驱动。本书中所介绍的设备驱动均指内核态驱动。

13.2 内核模块编程

本节的内容是介绍内核模块编程，介绍时首先直接给出完整例程并逐步分析的方法，然后介绍内核模块的编译和测试方法。作为提高内容，本节还介绍了模块参数的用法。

13.2.1 编写源码

内核模块的编程必须依照固定的模式，以下是一个简单的内核模块的例子：

```
#include <linux/module.h> /* 内核模块编程所需的头文件 */

MODULE_LICENSE("GPL"); /* 版权声明 */

/* 模块的初始化函数 */
static __init int demo_init(void)
{
    printk(KERN_ALERT "demo_init: be called.\n");
    return 0;
}

/* 模块的退出函数 */
static __exit void demo_exit(void)
{
    printk(KERN_ALERT "demo_exit: be called.\n");
}

module_init(demo_init); /* 指定 demo_init 为初始化函数 */
module_exit(demo_exit); /* 指定 demo_exit 为退出函数 */
```

下面将对源码中的各部分进行解释。

一、头文件

```
#include <linux/module.h>
```

这一行包含了内核模块编程所必需的头文件。注意，编译内核模块时头文件不是在系统头文件目录中搜索，而是在内核源码的 `include` 目录中搜索。

二、版权声明

```
MODULE_LICENSE("GPL");
```

这一行将代码的版权声明为 GPL。因为 GNU/Linux 内核本身是 GPL 版权。内核模块被认为是在内核基础上进行的二次开发，因此也必须遵循 GPL 版权。如果模块中没有版权声明，则加载时可能会被内核拒绝。版权声明放在源码的任何位置都可以。

三、初始化函数

```
static __init int demo_init(void);
```

这一行声明了模块的初始化函数。与应用程序不同，内核模块没有入口函数 `main`，而有一个初始化函数。初始化函数只在模块加载的时候被调用一次。其中的 `__init` 是可选的，它告诉编译器将此函数放在一个特殊的代码段上，当模块加载成功后，这个代码段就被释放掉以节约内存空间。

初始化函数必须有一个 `int` 型的返回值，返回 `0` 表示初始化成功，否则表示初始化失败。按惯例，内核的函数在失败时返回一个负数，代表失败的原因，称为错误码。内核的错误码都以宏的形式定义在头文件 `<linux/errno.h>` 中，这些宏定义都是正数，所以返回时要加一个负号。



在进行内核编程时，因为编译器的语法检查较为严格，如果函数没有参数，最好写上 `void`，否则会产生一个警告。

只进行如上的声明不足以表明 `demo_init` 就是模块的初始化函数，还必须加上以下一行源码：

```
module_init(demo_init);
```

`module_init` 是一个宏，它的作用是将所给的函数指定为模块的初始化函数。

四、退出函数

```
static __exit void demo_exit(void);
```

这一行声明了模块的退出函数。退出函数只在模块被卸载的时候调用一次。其中的 `__exit` 是可选的，它告诉编译器，如果模块被编译进内核，则这个函数根本不需要编译成目标代码，因为在这种情况下不存在卸载操作。

同样，只进行如上的函数声明不足以表明 `demo_exit` 就是模块的退出函数，还必须加上以下一行源码：

```
module_exit(demo_exit);
```

`module_exit` 也是一个宏，它的作用是将所给的函数指定为模块的退出函数。

有了这四个部分，一个内核模块就完整了。

13.2.2 printk 函数

13.2.2.1 用法

`printk` 是一个内核提供的用于输出信息的函数，用法类似于应用编程中的 `printf` 函数，其原型如下：

```
int printk(const char * fmt, ...);
```

与 `printf` 函数不同的是，它的格式字符串前面可以加一个表示级别的前缀，用尖括号包围的数字表示，如：

```
printk("<1>demo_init: num = %d, str = %s.\n", num, str);
```

级别的取值范围是从 `0` 到 `7`，在内核源码中，每个级别相应的定义如下：

```
#define KERN_EMERG      "<0>" /* 系统不可用 */
```

```
#define KERN_ALERT      "<1>" /* 必须立刻采取措施 */
#define KERN_CRIT       "<2>" /* 严重情况 */
#define KERN_ERR        "<3>" /* 错误 */
#define KERN_WARNING    "<4>" /* 警告 */
#define KERN_NOTICE     "<5>" /* 正常但需引起注意 */
#define KERN_INFO       "<6>" /* 显示信息 */
#define KERN_DEBUG      "<7>" /* 调试级别的信息 */
```

实际上, 内核中对 `printk` 函数的每个信息级别又包装了一个相应的函数, 它们的原型如下:

```
int pr_emerg(const char * fmt, ...); /* 对应 KERN_EMERG */
int pr_alert(const char * fmt, ...); /* 对应 KERN_ALERT */
int pr_crit(const char * fmt, ...); /* 对应 KERN_CRIT */
int pr_err(const char * fmt, ...); /* 对应 KERN_ERR */
int pr_warning(const char * fmt, ...); /* 对应 KERN_WARNING */
int pr_notice(const char * fmt, ...); /* 对应 KERN_NOTICE */
int pr_info(const char * fmt, ...); /* 对应 KERN_INFO */
int pr_debug(const char * fmt, ...); /* 对应 KERN_DEBUG */
int pr_devel(const char * fmt, ...); /* 基本与 pr_debug 相同 */
```

函数 `pr_devel` 比较特殊, 它只有在定义了宏 `DEBUG` 时才起作用, 否则就相当于一条空语句。这样当程序在调试时可以加上 `DEBUG` 宏定义使函数 `pr_devel` 有输出, 在正式发布时去掉 `DEBUG` 宏定义, 就等价于去掉了所有的 `pr_devel` 函数。函数 `pr_debug` 也有这样的作用。

13.2.2.2 查看输出信息

`printk` 函数输出的信息首先放在内核的一块缓冲区中, 然后级别小于一个固定值的信息会被输出到默认控制台上, 这个值可以通过 `proc` 文件系统查看:

```
cat /proc/sys/kernel/printk # 查看内核的信息输出级别
```

这个值也可以被修改:

```
echo 8 >/proc/sys/kernel/printk # 将内核的信息输出级别改为 8
```

修改时必须有 `root` 权限。将这个值改为 8 之后所有级别的信息都会被输出到默认控制台上。从图形界面启动的控制台都不是默认控制台, 因此不能直接看到 `printk` 函数输出的信息, 但可以用以下命令查看内核的信息缓冲区:

```
dmesg
```

由于内核在启动阶段就输出了大量的信息, 而调试时通常只关心最新输出的信息, 所以经常用 `tail` 命令加以过滤, 如:

```
dmesg | tail
```

用以下命令可将内核的信息缓冲区清空:

```
dmesg -c # 清空信息缓冲区需要 root 权限
```

13.2.3 编译内核模块

一般将内核模块的源码单独放在一个目录中,但要使用相应的内核源码中的 **Makefile** 进行编译。这时要求模块源码的目录中也有一个简单的 **Makefile** 以说明要编译哪些模块。假定模块的源码只有一个 **demo.c**,则这个 **Makefile** 可以这样写:

```
obj-m := demo.o
```

内核的 **Makefile** 将引用这个 **Makefile** 并根据 **obj-m** 变量的值去编译相应的模块,结果是将源码 **demo.c** 最终编译为内核模块文件 **demo.ko**。如果模块包含多个源文件,则可以这样写:

```
obj-m := demo.o
demo-objs := file1.o file2.o
```

其中 **obj-m** 变量的值仍然是要编译的模块,**demo-objs** 变量的值表示 **demo** 这个模块是由哪些目标文件链接而成的。最终产生的作用是将源码 **file1.c** 和 **file2.c** 编译成内核模块文件 **demo.ko**。这样写的话需要注意的是,**demo.c** 是不编译的,而且不能再把 **demo.o** 放到 **demo-objs** 变量中去,因此模块的名称(去掉 **.ko** 后缀)就不能与任何一个源文件的名称(去掉 **.c** 后缀)相同。

写好 **Makefile** 以后,可以用以下命令开始编译:

```
make -C kernel-dir M=$(pwd) modules
```

其中 **-C** 参数指定 **Makefile** 所在的目录;**M** 是一个变量,内核的 **Makefile** 将这个变量的值理解为模块源码所在的目录,这里把它设为了当前目录;**modules** 则是一个专门用于编译模块的目标。

大多数发行版上安装了内核编译环境后,内核源码在以下目录中:

```
/lib/modules/$(uname -r)/build
```

实际上这个目录中并不包含完整的源码,只包含编译模块所需的文件。

如果希望直接用 **make** 命令开始编译,则可以利用 **Makefile** 中的分支结构,写一个如下的 **Makefile**:

```
ifeq ($(KERNELRELEASE),) # 如果 KERNELRELEASE 无定义,说明是在当前目录
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build # 内核目录
    PWD := $(shell pwd) # 当前目录
default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules # MAKE 变量的默认值就是 make
clean:
    # 删除编译时产生的文件 #
else # 如果 KERNELRELEASE 已定义,说明是被内核的 Makefile 引用
    obj-m := demo.o
endif
```

在当前目录直接执行 **make** 命令时,**KERNELRELEASE** 变量无定义,因此实际执行的是 **ifeq** 到 **else** 之间的内容,这里定义了一个默认目标,再次执行 **make** 命令,使用的是内核的 **Makefile**。

当内核的 `Makefile` 根据 `M` 变量的值引用当前目录下的 `Makefile` 时, `KERNELRELEASE` 变量有了值, 实际起作用的是 `else` 和 `endif` 之间的部分, 这样就正确地完成了整个编译过程。理解这个 `Makefile` 的关键就是它被使用了两次, 而这两次起的作用是不同的。

13.2.4 加载与卸载

使用 `insmod` 命令可以加载内核模块, 如:

```
insmod demo.ko # 加载内核模块需要 root 权限
```

加载时将调用模块的初始化函数。实际上, 初始化函数就执行在 `insmod` 进程的上下文内。用 `lsmod` 命令可以查看所有已经加载的内核模块:

```
lsmod # 查看已经加载的内核模块
```

这条命令除显示模块的名字之外, 还显示模块的大小、引用计数等信息。还可以通过查看 `proc` 文件系统中的 `/proc/modules` 文件得到已加载内核模块的更多信息:

```
cat /proc/modules # 查看已加载的内核模块
```

用 `rmmod` 命令可以卸载内核模块, 如:

```
rmmod demo.ko # 卸载内核模块需要 root 权限
```

卸载时将调用模块的退出函数。实际上, 退出函数就执行在 `rmmod` 进程的上下文内。

13.2.5 模块参数

13.2.5.1 基本用法

内核模块支持模块参数的技术, 在加载模块时, 可以在命令行上指定模块中全局变量的初值。作为参数的全局变量必须在源码中有额外的说明, 如:

```
static int num = 0; /* 定义全局变量 num */
module_param(num, int, S_IRUGO); /* 将 num 指定为整型的模块参数 */
static char *str = ""; /* 定义全局字符指针 str */
module_param(str, charp, S_IRUGO); /* 将 str 指定为字符串型的模块参数 */
```

宏 `module_param` 用于指定模块参数, 它的第一个参数是指定的变量名, 也是加载时所使用的参数名称, 第二个参数是模块参数的类型, 第三个参数表示访问权限。

定义了模块参数后, 就可以在加载模块时指定它们的初值, 如:

```
insmod demo.ko num=5 str=ABCD # 指定 num 的初值为 5, str 的初值为 "ABCD"
```

这些初值在初始化函数被调用之前就已经设置好了。

模块参数的类型并不与 C 语言的变量类型一一对应, 它们之间的对应关系如表 13.2 所示。

表 13.2 模块参数与变量类型的对应关系

参数类型	变量类型	说明
bool	int	布尔型，指定值为 0 时，变量初值为 0；指定值非 0 时，变量初值为 1
invbool	int	反布尔型，指定值为 0 时，变量初值为 1；指定值非 0 时，变量初值为 0
charp	char *	字符串型，变量指向一个字符串，字符串内容为指定的内容
byte	char	字节型，变量初值等于指定值
int	int	整型，变量初值等于指定值
short	short	短整型，变量初值等于指定值
long	long	长整型，变量初值等于指定值
uint	unsigned int	无符号整型，变量初值等于指定值
ushort	unsigned short	无符号短整型，变量初值等于指定值
ulong	unsigned long	无符号长整型，变量初值等于指定值

内核有一个 `sysfs` 文件系统，这个文件系统挂载在 `/sys` 目录下。对应于每一个已加载的内核模块，在 `/sys/module` 目录下都有一个同名的目录，里面存放的是关于这个模块的一些信息。其中有一个目录 `parameters`，里面的每个文件都代表着一个模块参数，文件的访问权限实际上就是由 `module_param` 指定的。例如，在 `demo` 模块中按上述代码定义模块参数，加载模块后输入以下命令：

```
ls -l /sys/module/demo/parameters/
```

将显示以下内容：

```
-r--r--r-- 1 root root 4096 06-21 14:51 num
-r--r--r-- 1 root root 4096 06-21 14:51 str
```

`S_IRUGO` 是所有用户只读的权限，因此产生的文件就是这个权限的。如果将权限设置为可写的，则还可以通过这个文件修改内核模块中变量的值，但修改后内核没有主动通知到模块的机制，因此必须由模块自己检测才能发现变量值被修改了。

13.2.5.2 命名参数

如果希望指定参数的名称，而不是与所定义的变量同名（比如变量是一个结构体的成员时），则可以使用如下代码：

```
struct param {
    int num;
    char *str;
} param = {0, ""}; /* 定义变量以存放参数 */
module_param_named(num, param.num, int, S_IRUGO); /* num 对应 param.num */
module_param_named(str, param.str, charp, S_IRUGO); /* str 对应 param.str */
```

以上代码实际上定义了两个参数：`num` 和 `str`，通过 `num` 参数指定的值存放到结构体变量 `param` 的 `num` 成员中，通过 `str` 参数指定的值存放到结构体变量 `param` 的 `str` 成员中。与 `module_param` 宏相比，`module_param_named` 宏多出的第一个参数表示模块参数的名称，其他

参数的含义相同。

13.2.5.3 数组参数

模块还支持数组参数，例如：

```
static char ch[5] = {'\0', '\0', '\0', '\0', '\0'}; /* 定义数组 */
static int nr_ch = 0; /* 整型变量用于存放元素个数 */
module_param_array(ch, byte, &nr_ch, S_IRUGO); /* 指定 ch 为模块参数 */
```

`module_param_array` 的第一个参数是数组名，也是加载时所使用的参数名称，第二个参数是类型，这个类型应与数组元素的类型相对应，第三个参数是一个全局整型变量的地址，用于存放加载时所提供的数据个数，第四个参数是访问权限。

由以上代码定义数组形式的模块参数后，加载时可用以下命令指定数组元素的值：

```
insmod demo.ko ch=0x31,0x32,0x33,0x34,0x35
```

这样加载后，`ch` 数组的各个元素的初值分别设为对应的值，`nr_ch` 的值设为所提供的初值的个数。如果提供的值的个数超过数组的大小，加载将失败。

同样有一个数组参数的命名版本，可以这样使用：

```
module_param_array_named(char, ch, byte, &nr_ch, S_IRUGO);
```

这条语句将参数的名称指定为 `char`，其余部分与 `module_param_array` 的参数相同。



参数的名称是在加载模块的命令行上使用的，在模块源代码里，它实际上是一个字符串，因此与 C 语言关键字不冲突。

13.3 字符设备驱动

字符设备是 Linux 系统上一种最基本的设备类型。本节的主要内容是介绍字符设备的相关概念和编程接口，最后通过一个例程来说明字符设备的常用编程模式。

13.3.1 设备文件与设备号

在 Linux 系统上，多数设备被抽象为设备文件供应用程序使用。设备文件支持和普通文件一样的打开、关闭、读写等系统调用。这种访问接口的统一使很多应用程序可以同时支持对普通文件和设备文件的操作，无须区别对待。

这种通过设备文件访问的设备可分为两类：字符设备和块设备。字符设备的特点是支持基本的打开、关闭、读和写等操作，一般没有读写位置的概念，数据的输入输出只有时间上的顺序。块设备虽然也支持这些基本操作，但它最大的不同是，数据的读写以块为单位，块的大小是统一的，并且有存储位置的概念，可以随机访问指定位置上的数据。如果块设备有足够的容量，就可以容纳一个文件系统，有了文件系统的块设备就可以被挂载。

系统中的磁盘一般被抽象为块设备，其他硬件设备（如串口、键盘、鼠标等）抽象为字符设备。需要注意的是，字符设备和块设备只是一种抽象的概念，它可以不对应于任何具体硬件设备，一个

硬件设备也可能同时抽象为几个字符设备和块设备。

设备文件在文件系统中有一个对应的索引节点，但并不占用存储文件内容的空间。它与内核的设备驱动通过设备号来联系。设备号可分为两个部分：主设备号和次设备号。

用 `mknod` 命令可以建立设备文件，如：

```
mknod -m 666 test c 5 0 # 建立设备文件 test, 需 root 权限
```

其中 `-m` 参数用于指定文件的权限，`666` 是权限标志的八进制形式，`test` 是设备文件的名称，`c` 表示字符设备，`5` 和 `0` 分别是主设备号和次设备号。

设备类型、主设备号、次设备号决定了唯一的设备，如果两个设备文件的这三个属性都相同，它们实际上就对应着同一个设备。设备文件的设备号可以任意设置，甚至指向内核中并不存在的设备。传统上，同一主设备号表示同一种设备，由同一个设备驱动程序驱动；而同一主设备号下的不同次设备号则表示同一种设备的多个实体。

用 `ls -l` 命令可以查看设备文件的详细信息，例如对于上面建立的设备文件 `test`，显示信息如下：

```
crw-rw-rw- 1 root root 5, 0 06-22 18:09 test
```

第一列的 `c` 表示文件的类型为字符设备文件，对普通文件来说显示文件长度的位置上实际显示的是主设备号和次设备号。

13.3.2 字符设备编程接口

13.3.2.1 设备号的注册与注销

每个字符设备驱动都至少要占用一个设备号，而设备号在使用之前必须向内核注册。设备号被某个驱动注册之后就表示它已经被占用了，其他驱动试图注册同一设备号的操作将失败。可以将设备号看成一种有限的资源，内核对此加以管理以防止多个驱动用同一设备号而产生冲突。与设备号管理有关的函数接口都声明在以下头文件中：

```
#include <linux/fs.h>
```

一、注册设备号

注册指定设备号的接口函数原型如下：

```
int register_chrdev_region(dev_t from, unsigned int count, const char *name);
```

其各个参数及返回值的含义解释如下。

- ◆ `from`：表示要注册的设备号的起始值。
- ◆ `count`：表示要注册的设备号的个数。
- ◆ `name`：字符串，设备的名称。
- ◆ 返回值：`0` 表示成功，负数为错误码。

在驱动中可以用此接口同时注册由 `from` 开始的 `count` 个设备号，即由连续的设备号组成的一段区域。`dev_t` 是内核用来表示设备号的数据类型，定义如下：

```
typedef unsigned int __u32;
typedef __u32 __kernel_dev_t;
typedef __kernel_dev_t dev_t;
```

可见 `dev_t` 实质上就是 32 位的无符号整型数，内核用它的高 12 位表示主设备号，低 20 位表示次设备号。内核还提供了两个接口，专门用于从 `dev_t` 型的数据得到主次设备号：

```
unsigned int MAJOR(dev_t dev); /* 返回 dev 的主设备号 */
unsigned int MINOR(dev_t dev); /* 返回 dev 的次设备号 */
```

还有一个接口可以从主次设备号构造出 `dev_t` 型数据：

```
dev_t MKDEV(unsigned int major, unsigned int minor);
```

其中参数 `major` 是主设备号，参数 `minor` 是次设备号，返回值是构造出的设备号。这些接口都是由宏实现的。



驱动编程有一个原则，如果内核有相应的接口，则尽量不要涉及内核的实现细节，这样可以给驱动带来对于不同内核版本的更好的可移植性。

查看文件 `/proc/devices` 可得到所有设备的名称及所占用主设备号，这里的设备名称就是由注册设备号时的 `name` 参数提供的。

二、动态分配设备号

注册设备号时，如果所指定的设备号已被注册，则注册将失败。在很多情况下并不需要为设备明确指定一个固定的设备号，这时可以让内核自动分配设备号，接口函数原型如下：

```
int alloc_chrdev_region(dev_t *dev, unsigned int baseminor, unsigned int count,
    const char *name);
```

其各个参数及返回值的含义解释如下。

- ◆ `dev`：输出型参数，用于存放分配到的设备号。
- ◆ `baseminor`：要注册的次设备号的起始值。
- ◆ `count`：要注册的设备号的个数。
- ◆ `name`：字符串，表示设备的名称。
- ◆ 返回值：0 表示成功，负数为错误码。

分配设备号时，由内核自动决定的实际上是主设备号，所需的次设备号的起始值和个数还是由调用者提供。最终由 `dev` 参数返回的是分配到的第一个设备号。

分配到的设备号为已注册状态，不需要再调用注册设备号的接口函数进行注册。

三、注销设备号

当绑定在某个设备号上的设备已经不存在，或者驱动模块要退出时，所注册的设备号需要进行注销。设备号被注销表示它又处于空闲状态，可以被再次注册。注销设备号的接口函数原型如下：

```
void unregister_chrdev_region(dev_t from, unsigned int count);
```

其各个参数的含义解释如下。

- ◆ **from**: 要注销的设备号的起始值。
- ◆ **count**: 要注销的设备号的个数。

这里需要注意的是, 注销与注册操作必须是一一对应的, 一次注册得到的设备号区域不能分多次注销。注销时指定的设备号起始值及个数必须等于注册时的值, 这里说的注册包括自动分配设备号。

13.3.2.2 字符设备的注册与注销

内核用一个结构体表示字符设备。结构体在使用之前其成员必须初始化, 然后将其向内核注册, 这样内核中就多了一个字符设备, 相应地有字符设备的注销操作。相关的接口函数都声明在以下头文件中:

```
#include <linux/cdev.h>
```

一、初始化字符设备

初始化字符设备的接口函数原型如下:

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops);
```

其各个参数的含义解释如下。

- ◆ **cdev**: 指向被初始化的字符设备。
- ◆ **fops**: 指向字符设备支持的一系列文件操作。

初始化字符设备实际上就是对 **cdev** 参数所指向的结构体中的成员进行初始化。结构体 **cdev** 的定义如下:

```
struct cdev {  
    struct kobject kobj; /* 内核对象, 内部使用 */  
    struct module *owner; /* 指向字符设备的所属模块 */  
    const struct file_operations *ops; /* 指向文件操作 */  
    struct list_head list; /* 链表节点, 内部使用 */  
    dev_t dev; /* 绑定的设备号起始值, 内部使用 */  
    unsigned int count; /* 绑定的设备号的个数, 内部使用 */  
};
```

其中标有“内部使用”的成员为内核管理字符设备所使用的成员, 在驱动中一般不要直接对其进行修改。

owner 成员指向一个 **struct module** 型的结构体, 这个结构体是用于管理内核模块的, 对应于每一个加载的模块都有一份相应的数据。**owner** 成员的主要用途是正确处理模块的引用计数, 当字符设备被打开使用时, 它所属模块的引用计数将增加, 而引用计数为 0 的模块才能被卸载, 这样就避免了将正在使用的设备驱动卸载的情况。初始化字符设备的函数并没有设置这个成员, 需要另外赋值。一般情况下都会赋值为 **THIS_MODULE**, 这是一个宏, 代表本模块。

二、创建字符设备

通常将动态分配数据结构并初始化的操作称为“创建”。内核也提供了一个创建字符设备的接口函数，其原型如下：

```
struct cdev *cdev_alloc(void);
```

它的返回值指向新创建的字符设备结构体，返回 `NULL` 则表示创建失败。

这个接口函数将动态分配一块内存，并按 `struct cdev` 结构体将其初始化，然后返回指向它的指针。需要指出的是，调用这个函数时并没有提供指向文件操作的指针，因此，通过这种方式得到的字符设备结构体的 `ops` 成员还需要另外赋值。

另外需要注意的一点是，创建时动态分配的内存会在字符设备被注销时自动释放。

三、注册字符设备

初始化或创建得到的字符设备还仅仅是一个结构体变量，要使内核中真正产生一个字符设备，需要将其注册。注册字符设备的接口函数原型如下：

```
int cdev_add(struct cdev *p, dev_t dev, unsigned count);
```

其各个参数及返回值的含义解释如下。

- ◆ `p`：指向要注册的字符设备。
- ◆ `dev`：此字符设备要绑定的设备号的起始值。
- ◆ `count`：此字符设备要绑定的设备号的个数。
- ◆ 返回值：0 表示注册成功，负数表示错误码。

注册以后，这个字符设备就真正起作用了。在设备存在期间，参数 `p` 指向的结构体可能会被内核访问，因此它必须一直存在。

注册字符设备时，可以将它与一段设备号的区域绑定。应用程序访问字符设备文件时，实际上就是在访问与设备文件的设备号绑定的字符设备。

四、注销字符设备

注销是注册的反操作。注销一个字符设备后，它就不再起作用，相当于内核中没有了这个设备。注销字符设备的接口函数原型如下：

```
void cdev_del(struct cdev *p);
```

其中参数 `p` 指向被注销的字符设备。

注销时，如果参数 `p` 指向的字符设备原来是用 `cdev_alloc` 函数创建的，则其存储空间同时被释放；如果仅仅是用 `cdev_init` 初始化过，则不会被释放。实际上，关于如何“释放”一个字符设备的方法记录在它的 `kobj` 成员中，初始化和创建字符设备时，对这个成员的初始化方法是不同的。

五、同时注册设备号与字符设备

内核提供了另外一个接口函数，它可以在注册设备号的同时注册字符设备，其原型如下：

```
int register_chrdev(unsigned int major, const char *name,  
    const struct file_operations *fops);
```

其各个参数及返回值的含义解释如下。

- ◆ **major**: 要注册的主设备号, 如果传入 0 则内核自动分配设备号。
- ◆ **name**: 设备的名称。
- ◆ **fops**: 指向字符设备所支持的一系列文件操作。
- ◆ **返回值**: 正数表示自动分配得到的设备号, 0 表示注册成功, 负数表示错误码。

使用这个接口函数时, 只需要提供 (或自动分配) 一个主设备号。实际上它将注册这个主设备号下从 0 到 255 的次设备号区域。这个函数是为了保持向下兼容而提供的, 在它内部其实就使用了前述几个函数, 所注册的字符设备是动态创建的。

六、同时注销设备号与字符设备

与同时注册设备号及字符设备的接口函数相对应, 内核也提供了一个同时注销设备号及字符设备的接口函数, 原型如下:

```
void unregister_chrdev(unsigned int major, const char *name);
```

其各个参数的含义解释如下。

- ◆ **major**: 要注销的主设备号。
- ◆ **name**: 设备的名称, 实际上没有用。

13.3.3 文件操作

要注册一个字符设备, 驱动必须提供所对应的一系列文件操作。这些操作由以下结构体表示:

```
struct file_operations {  
    struct module *owner; /* 指向文件操作的所属模块, 一般为 THIS_MODULE */  
    /* 各种文件操作 */  
    loff_t (*llseek)(struct file *, loff_t, int);  
    ssize_t (*read)(struct file *, char __user *, size_t, loff_t *);  
    ssize_t (*write)(struct file *, const char __user *, size_t, loff_t *);  
    int (*open)(struct inode *, struct file *);  
    int (*release)(struct inode *, struct file *);  
    /* 这里只写出了部分成员 */  
};
```

这个结构体除了成员 **owner** 外, 其他成员都是函数指针, 这些指针所指向的函数由驱动提供。并非所有的函数都是必需的, 如果驱动不提供某个函数, 则相应的函数指针应赋值为 **NULL**。字符设备注册以后, 这些函数将由内核调用, 以实现设备的各种功能。

下面将介绍设备的几个基本文件操作: 打开、关闭、读、写和定位。

一、打开操作

当应用程序打开设备文件时, 内核将调用相应设备的打开操作函数, 它的函数指针定义如下:

```
int (*open)(struct inode *inode, struct file *file);
```

其各个参数及返回值的含义解释如下。

- ◆ `inode`: 指向被打开的索引节点。
- ◆ `file`: 指向打开后的文件。
- ◆ 返回值: 0 表示打开成功, 负数代表错误码。

当内核调用打开操作函数时, 会将代表被打开索引节点的结构体指针通过参数 `inode` 传进来, 这个结构体的类型定义如下:

```
struct inode {
    dev_t i_rdev; /* 索引节点对应的设备号 */
    union {
        struct pipe_inode_info *i_pipe; /* 指向代表命名管道的结构体 */
        struct block_device *i_bdev; /* 指向代表块设备的结构体 */
        struct cdev *i_cdev; /* 指向代表字符设备的结构体 */
    };
    /* 这里只写出了部分成员 */
};
```

其中的 `i_rdev` 成员提供了索引节点对应的设备号, 但通常不直接访问这个成员, 而是通过以下接口函数获得索引节点对应的主次设备号:

```
unsigned int imajor(const struct inode *inode); /* 获得主设备号 */
unsigned int iminor(const struct inode *inode); /* 获得次设备号 */
```

`struct inode` 类型包含一个内嵌的联合体。如果被打开的索引节点对应着一个字符设备, 则成员 `i_cdev` 是有效的, 它指向内核中对应的字符设备。



驱动可以访问 `inode` 参数指向的这些成员, 但不能修改它们。

内核还会把代表已打开的文件的结构体指针通过参数 `file` 传进来, 这个结构体的类型定义如下:

```
struct file {
    const struct file_operations *f_op; /* 指向文件对应的文件操作 */
    unsigned int f_flags; /* 文件的标志位 */
    loff_t f_pos; /* 文件的读写位置 */
    void *private_data; /* 内核不用, 由驱动使用, 指向的数据称为驱动的私有数据 */
    /* 这里只写出了部分成员 */
};
```

成员 `f_op` 是被打开的文件对应的操作。值得注意的是, 它的值可以由驱动在 `open` 函数里安全地修改, 所做的修改在 `open` 函数返回后生效。这可以被用来实现以下的功能: 注册一个字符设备, 绑定在多个设备号上, 这时只提供一个打开操作, 设备的具体功能在打开时根据设备号来决定。内核本身多处使用了这种技术, 如 `misc` 设备。

成员 `f_flags` 是文件的各种标志。标志位定义在头文件 `<linux/fcntl.h>` 中。应用程序可在 `open` 和 `fcntl` 系统调用时设置这些标志。驱动一般只会关心其中的 `O_NONBLOCK` 标志, 它表示

文件是否以非阻塞方式操作。

成员 `f_pos` 是文件的读写位置，它是一个 `loff_t` 型数据，这个类型是这样定义的：

```
typedef long long __kernel_loff_t;
typedef __kernel_loff_t loff_t;
```

因此它实际上是一个 64 位的超长整型数据。驱动可以对它进行读写。不过很多字符设备并没有读写位置的概念，可以不必处理。

成员 `private_data` 的用法比较特殊，内核本身不会读写这个成员，而是预留给驱动使用的。一般来说，驱动要为每个设备建立一些变量，用以保存一些必要的的数据，称为设备的私有数据。这些数据可以包装成一个结构体，然后把结构体的指针赋给 `private_data` 成员，这样就把设备数据和打开的文件联系到了一起。

打开操作不是必须提供的，如果驱动不提供，则对设备的打开操作默认总是成功的。

二、关闭操作

当应用程序关闭已打开的设备文件时，内核将调用相应设备的关闭操作函数，它的函数指针定义如下：

```
int (*release)(struct inode *inode, struct file *file);
```

其各个参数及返回值的含义解释如下。

- ◆ `inode`：指向要关闭的索引节点。
- ◆ `file`：指向要关闭的文件。
- ◆ 返回值：0 表示关闭成功，负数代表错误码。

关闭操作逻辑上是打开操作的反操作，如果打开时进行过需要清理的操作，如注册系统资源、分配内存等，一般的做法是在关闭操作里按相反的顺序进行清理。

三、读写操作

当应用程序从已打开的设备文件读取数据时，内核将调用相应设备的读操作函数，它的函数指针定义如下：

```
ssize_t (*read)(struct file *file, char __user *buf, size_t count,
                loff_t *pos);
```

其各个参数及返回值的含义解释如下。

- ◆ `file`：指向要读取的文件。
- ◆ `buf`：指向数据缓冲区，给应用程序的数据应放在这里。这是一个用户态内存指针。
- ◆ `count`：数据缓冲区的大小，单位为字节，给应用程序的数据不得多于此数。
- ◆ `pos`：指向文件的读写位置。
- ◆ 返回值：正数为实际读到的字节数，0 表示文件已结束，负数表示错误码。

当应用程序向已打开的设备文件写入数据时，内核将调用相应设备的写操作函数，它的函数指针定义如下：


```
ssize_t (*write)(struct file *file, const char __user *buf, size_t count,
                 loff_t *pos);
```

其各个参数及返回值的含义解释如下。

- ◆ file: 指向要写入的文件。
- ◆ buf: 指向数据缓冲区, 存放应用程序要写入的数据。这是一个用户态内存指针。
- ◆ count: 应用程序要写入的数据的字节数。
- ◆ pos: 指向文件的读写位置。
- ◆ 返回值: 正数或 0 为实际写入的字节数, 负数表示错误码。

当内核调用读操作函数时, 驱动要将返回给应用程序的数据放在 **buf** 所指的数据缓冲区中, 函数的返回值则为数据的字节数。当内核调用写操作函数时, 驱动可从 **buf** 所指的数据缓冲区中得到要写入的数据, 数据的字节数则由参数 **count** 指明。注意 **buf** 是一个用户态内存指针, 内核对这种指针通常加上 **__user** 标识, 它的值虽然也是内存地址, 但并不是内核的虚拟地址, 因此在驱动中不能直接访问。

读写操作函数的返回值是实际读写的数据的字节数, 它并不是必须与参数 **count** 相等的, 驱动应该根据设备的实际情况读写适量的数据。



一般应避免写操作返回 0, 因为很多应用程序在这种情况下会不断尝试重写, 导致死循环。当设备无法写入时, 应返回相应的错误码。

ssize_t 和 **size_t** 类型是专门用于表示各种“大小”的类型, 它们的定义如下:

```
typedef unsigned int __kernel_size_t;
typedef int __kernel_ssize_t;
typedef __kernel_size_t size_t;
typedef __kernel_ssize_t ssize_t;
```

可见它们都是整型, 唯一的区别是 **ssize_t** 是有符号的, **size_t** 是无符号的。

参数 **pos** 是指向文件读写位置的指针, 需要注意的是, 它并不指向 **file->f_pos**, 而是指向内核中的一个局部变量。内核在调用读操作函数前, 将 **file->f_pos** 的值读到这个变量中, 调用结束后再写回 **file->f_pos**。因此, 如果要在读写操作中修改文件的读写位置, 直接修改 **file->f_pos** 是无效的, 必须修改 **pos** 参数指向的变量。

四、定位操作

当应用程序使用 **lseek** 系统调用修改文件的读写位置时, 内核将调用定位操作函数, 它的函数指针定义如下:

```
loff_t (*llseek)(struct file *file, loff_t offset, int whence);
```

其各个参数及返回值的含义解释如下。

- ◆ file: 指向要操作的文件。

- ◆ **offset**: 要修改的偏移量。
- ◆ **whence**: 偏移量的起点。
- ◆ **返回值**: 0 或正数表示新的读写位置, 负数表示错误码。

其中 **whence** 参数有以下三个可选值。

- ◆ **SEEK_SET**: 相对于文件的开始。
- ◆ **SEEK_CUR**: 相对于当前读写位置。
- ◆ **SEEK_END**: 相对于文件的末尾。

在定位操作函数里, 驱动应直接更新 **file->f_pos** 的值, 并返回更新以后的值。定位操作不是驱动必须实现的操作。但如果驱动不提供, 内核将提供一个默认的定位操作函数, 可能与预期结果不符, 因此最好由驱动实现。对于没有读写位置概念的设备来说, 这个函数可以直接返回错误 **-ESPIPE**, 或者将 **file->f_pos** 置 0 并返回 0。

13.3.4 访问用户态内存

在文件的读写操作里, 内核传入的缓冲区指针是用户态内存指针, 不能直接访问。对此, 内核提供了一些接口函数, 可以在用户态内存和内核内存间复制数据, 这些函数的声明在以下头文件中:

```
#include <linux/uaccess.h>
```

从用户态内存复制数据到内核内存的接口函数原型如下:

```
unsigned long copy_from_user(void *to,  
                             const void __user *from, unsigned long n);
```

其各个参数及返回值的含义解释如下。

- ◆ **to**: 指向目标内存。
- ◆ **from**: 指向源内存, 这是一个用户态内存指针。
- ◆ **n**: 要复制的字节数。
- ◆ **返回值**: 尚未复制的字节数。

从内核内存复制数据到用户态内存的接口函数原型如下:

```
unsigned long copy_to_user(void __user *to,  
                           const void *from, unsigned long n);
```

其各个参数及返回值的含义解释如下。

- ◆ **to**: 指向目标内存, 这是一个用户态内存指针。
- ◆ **from**: 指向源内存。
- ◆ **n**: 要复制的字节数。
- ◆ **返回值**: 尚未复制的字节数。

这两个函数的返回值都是没有成功复制的字节数，因此驱动中常用返回值是否为 0 作为复制操作是否成功的判断条件。

13.3.5 动态分配内存

类似于应用编程的 `malloc` 接口，内核中也提供了一个动态分配内存的接口函数，其原型如下：

```
void *kmalloc(size_t size, gfp_t flags);
```

其各个参数及返回值的含义解释如下。

- ◆ `size`：要分配的内存的大小，单位是字节。
- ◆ `flags`：分配内存的标志，驱动中常用的两个值是 `GFP_KERNEL` 和 `GFP_ATOMIC`。
- ◆ 返回值：指向分配得到的内存，如果分配失败则返回 `NULL`。

使用 `GFP_KERNEL` 标志调用 `kmalloc` 时，如果所需的内存暂时不能满足，则会使当前进程进入睡眠状态，以等待内核重新获得可用内存，因此这种方式只能用在进程上下文中；使用 `GFP_ATOMIC` 标志调用时，会立刻从内核的“应急”内存中进行分配，不会导致进程睡眠，因此可用在任何上下文中。

另外还有一个类似的动态分配内存的接口函数，它与 `kmalloc` 的区别在于分配得到的内存区域将被清零，其原型如下：

```
void *kzalloc(size_t size, gfp_t flags);
```

其各个参数及返回值的含义均与 `kmalloc` 函数相同。



使用 `kmalloc` 和 `kzalloc` 函数时需要注意，它们能够分配到的内存大小存在上限，一般不能超过 128KB。

与它们对应的释放内存的接口函数原型如下：

```
void kfree(const void *objp);
```

其中参数 `objp` 指向要释放的内存，它必须是由 `kmalloc` 或 `kzalloc` 分配得到的。有时为了安全的原因会用下面的函数释放内存：

```
void kzfree(const void *p);
```

这个函数在释放内存前会先将其全部清零，这样就可以防止其他的内核模块“窃取”内存中的残留内容。

13.3.6 内存操作

内核提供了很多与应用编程接口类似的内存和字符串操作函数，它们的声明在以下头文件中：

```
#include <linux/string.h>
```

虽然其中很多函数的用法和功能均与应用编程时所用的同名函数相同，但必须区分清楚：应用

编程时使用的函数都链接到 C 标准库，而内核编程时所用的函数都是内核自身的代码。如果目标平台对这些函数有特殊的实现，则使用目标平台所提供的代码，否则内核将提供默认的实现。

这里将对其中一些较为常用的函数进行介绍。

13.3.6.1 内存类

一、设置内存区域

设置内存区域的值可以用下面这个函数：

```
void *memset(void *s, int c, size_t count);
```

其各个参数及返回值的含义解释如下。

- ◆ s：指向要设置的内存区域。
- ◆ c：要设置的值。
- ◆ count：要设置的内存区域的长度，单位是字节。
- ◆ 返回值：等于 s。

这里要注意，memset 函数将把指定内存区域中的每个字节的值设为指定值，如果指定的值超出了一个字节的取值范围，原则上只取其低位字节。

二、复制内存区域

复制内存区域的值可以用下面这个函数：

```
void *memcpy(void *dest, const void *src, size_t count);
```

其各个参数及返回值的含义解释如下。

- ◆ dest：目标地址，指向数据写入的内存区域。
- ◆ src：源地址，指向读取数据的内存区域。
- ◆ count：要复制的内存区域的长度，单位是字节。
- ◆ 返回值：等于 dest。

这里要注意，src 和 dest 指向的内存区域不可重叠。如果要在目标和源内存区域有重叠的情况下进行复制，则要用下面这个函数：

```
void *memmove(void *dest, const void *src, size_t count);
```

其各个参数及返回值的含义与 memcpy 函数相同。

三、查找内存区域

在内存区域中查找指定字符可以用下面这个函数：

```
void *memchr(const void *addr, int c, size_t n);
```

其各个参数及返回值的含义解释如下。

- ◆ addr：指向要查找的内存区域。

- ◆ c: 要查找的字符。
- ◆ n: 要查找的内存区域的长度, 单位是字节。
- ◆ 返回值: 指向找到的字符, 如果没找到, 则是 `NULL`。

进行查找操作时, 将从指定内存区域的第一个字节开始向后查找, 找到第一个字节后查找停止。下面这个函数也可以在内存区域中查找指定字符:

```
void *memscan(void *addr, int c, size_t n);
```

其各个参数及返回值的含义与 `memchr` 函数相同, 功能也基本相同, 唯一的不同在于, 如果没找到指定的字符, 它的返回值指向内存区域后面的第一个字节, 而不是 `NULL`。

四、内存区域比较

下面这个函数可以比较两块内存区域的大小:

```
int memcmp(const void *cs, const void *ct, size_t count);
```

其各个参数及返回值的含义解释如下。

- ◆ cs: 指向被比较的源内存区域。
- ◆ ct: 指向被比较的目标内存区域。
- ◆ count: 被比较的内存区域的长度, 单位是字节。
- ◆ 返回值: 正数表示源内存区域大于目标内存区域, 0 表示两者相等, 负数表示源内存区域小于目标内存区域。

比较操作将从内存区域的第一个字节开始, 如果第一个字节相同, 则继续比较后面的字节。

13.3.6.2 字符串类

一、字符串复制类

下面的函数可以复制字符串:

```
char *strcpy(char *dest, const char *src);
```

其各个参数及返回值的含义解释如下。

- ◆ dest: 目标字符串。
- ◆ src: 源字符串。
- ◆ 返回值: 等于 dest。

注意参数 `dest` 指向的可用空间必须大于 `src` 字符串的长度。

下面这个函数也可以复制字符串, 并且可以限制复制的长度:

```
char *strncpy(char *dest, const char *src, size_t count);
```

其中参数 `count` 是复制的长度上限, 单位是字节, 其他参数及返回值的含义与 `strcpy` 函数相同。如果 `src` 字符串的长度小于上限 `count`, 参数 `dest` 指向的内存区域中将会被补写若干个字符 `\0`, 以使写入的总字节数达到 `count`; 如果 `src` 字符串的长度大于上限 `count`, 则只复制

count 个字节，dest 将不再是以 \0 结尾的字符串。

下面这个函数的功能与 strncpy 类似，但它可以保证复制后的字符串有结束符：

```
size_t strlcpy(char *dest, const char *src, size_t size);
```

其中参数 size 是复制的长度上限，单位是字节，其他参数及返回值的含义与 strcpy 函数相同。与 strncpy 不同的是，它不对参数 dest 指向的内存区域中补写字符 \0，如果 src 字符串的长度大于上限 size，则只复制前 size-1 个字符，并在 dest 内存末尾写入字符 \0。

二、字符串查找类

下面的函数可以在字符串中查找一个指定的字符：

```
char *strchr(const char *s, int c);
```

其各个参数及返回值的含义解释如下。

- ◆ s：被查找的字符串。
- ◆ c：要查找的字符。
- ◆ 返回值：指向找到的字符，如果未找到，则为 NULL。

进行查找操作时，将从字符串的第一个字节开始向后查找，找到第一个字节后查找停止。如果参数 s 指向的不是合法的字符串（不以字符 \0 结束），则有可能导致内存访问越界。

下面的函数也可以在字符串中查找指定字符，但查找方向是从后往前：

```
char *strrchr(const char *s, int c);
```

其各个参数及返回值的含义与 strchr 相同。使用这个函数进行查找操作时，将从字符串的最后一个字节开始向前查找，找到第一个字节后查找停止。

以上两个函数在查找时，如果被查找的字符串没有结束符，则很可能导致内存访问越界，而下面这个函数在查找时可以指定一个范围以避免这一问题：

```
char *strnchr(const char *s, size_t count, int c);
```

其中参数 count 是搜索范围的长度上限，单位是字节，其他参数及返回值的含义与函数 strchr 相同。这个函数的功能与 strchr 函数类似，只是查找时多了一个停止条件，即搜索的长度大于指定的上限 count。

下面的函数可以在字符串中查找子字符串：

```
char *strstr(const char *s1, const char *s2);
```

其各个参数及返回值的含义解释如下。

- ◆ s1：被查找的字符串。
- ◆ s2：要查找的子字符串。
- ◆ 返回值：找到的字符串的首地址，如果未找到，则为 NULL。

三、字符串连接类

下面的函数可以连接两个字符串：

```
char *strcat(char *dest, const char *src);
```

其各个参数及返回值的含义解释如下。

- ◆ dest: 目标字符串。
- ◆ src: 源字符串。
- ◆ 返回值: 等于 dest。

这个函数将把 src 字符串接续在 dest 字符串后面, 形成一个字符串。注意 dest 指向的内存区域的长度必须大于两字符串长度之和。

下面这个函数也可以连接两个字符串, 但对字符串的复制长度有限制:

```
char *strncat(char *dest, const char *src, size_t count);
```

其中参数 count 是复制长度的上限, 单位是字节, 其他参数及返回值的含义与 strcat 函数相同。这个函数与 strcat 函数的区别在于, 如果 src 字符串的长度大于 count, 则只复制前 count 个字符, 并在 dest 字符串最后加上 \0。

如果要限制连接以后的字符串的长度, 则要用下面这个函数:

```
size_t strlcat(char *dest, const char *src, size_t count);
```

其各个参数及返回值的含义解释如下。

- ◆ dest: 目标字符串。
- ◆ src: 源字符串。
- ◆ count: 连接后字符串的长度上限。
- ◆ 返回值: 字符串 dest 和 src 的长度之和。

注意它的返回值是两个字符串的长度之和, 而不是连接后字符串的长度。

四、字符串长度类

下面这个函数可以求出字符串的长度:

```
size_t strlen(const char *s);
```

其参数及返回值的含义解释如下。

- ◆ s: 目标字符串。
- ◆ 返回值: 字符串的长度。

注意如果字符串 s 没有以 \0 结尾, 则有可能导致内存访问越界。这时可以用下面这个函数, 它对扫描的长度有限制:

```
size_t strnlen(const char *s, size_t count);
```

其中参数 count 是扫描长度的限制, 其他参数及返回值的含义与 strlen 函数相同。这个函数在求字符串长度时, 扫描长度控制在 count 个字符以内, 也就是说, 返回的长度不可能超过 count。

五、字符串比较类

下面的函数可以比较两个字符串的大小：

```
int strcmp(const char *cs, const char *ct);
```

其各个参数及返回值的含义解释如下。

- ◆ cs：被比较的源字符串。
- ◆ ct：被比较的目标字符串。
- ◆ 返回值：正数表示源字符串大于目标字符串，0 表示两者相等，负数表示源字符串小于目标字符串。

如果要限制比较的长度，则可以用下面这个函数：

```
int strncmp(const char *cs, const char *ct, size_t count);
```

其中参数 count 是比较长度的上限，单位是字节，其他参数及返回值的含义与 strcmp 函数相同。这个函数在比较时，如果两个字符串的前 count 个字符都相同，则会返回相等，不再比较后面的字符。

下面的函数在比较两个字符串时不区分字母的大小写：

```
int strcasecmp(const char *s1, const char *s2);
```

其各个参数及返回值的含义与 strcmp 函数相同。这个函数在比较字符串时将同一个英文字母的大小写认为是相等的。

下面这个函数在比较两个字符串时，既有比较长度限制，又不区分字母的大小写：

```
int strncasecmp(const char *s1, const char *s2, size_t count);
```

其各个参数及返回值的含义与 strncmp 函数相同。

对以上内存和字符串操作的总结如表 13.3 所示。

表 13.3 内存和字符串操作

		内存操作	字符串操作	
			普通	有长度限制
设置		memset		
复制		memcpy, memmove	strcpy	strncpy, strlcpy
查找字符		memchr, memscan	strchr, strchr	strnchr
查找字符串			strstr	
连接			strcat	strncat, strlcat
求长度			strlen	strnlen
比较	区分大小写	memcmp	strcmp	strncmp
	忽略大小写		strcasecmp	strncasecmp

13.3.6.3 格式输出

内核也提供了类似应用编程的 `sprintf` 函数接口，可以向一个缓冲区格式化输出信息，它们都声明在 `<linux/kernel.h>` 头文件中。

向缓冲区格式化输出信息的接口函数原型如下：

```
int sprintf(char *buf, const char *fmt, ...);
```

其各个参数及返回值的含义解释如下。

- ◆ `buf`：缓冲区首地址。
- ◆ `fmt`：格式字符串。
- ◆ 返回值：构造出的目标字符串的长度。

这是一个有可变参数的函数，使用时注意要保证缓冲区大于目标字符串的长度，否则会引起内存访问越界。

下面这个函数同样可以向缓冲区格式化输出信息，并且对输出的长度有控制：

```
int snprintf(char *buf, size_t size, const char *fmt, ...);
```

其中参数 `size` 是输出信息的长度上限，其他参数及返回值的含义与 `sprintf` 函数相同。这个函数与 `sprintf` 函数类似，区别在于向缓冲区写入时，控制写入的字节数不超过 `size`。如果目标字符串过长，则将其截短并在末尾写上结束符 `\0`。一般把缓冲区的大小传递给参数 `size`，这样可以保证内存访问不越界。注意，即使目标字符串被截短，返回值仍是目标字符串的全长。

如果要得到写入缓冲区的字节数而不是目标字符串的长度，则要用以下函数：

```
int scnprintf(char *buf, size_t size, const char *fmt, ...);
```

其各个参数的含义与 `snprintf` 相同，而返回值是实际写入缓冲区的字节数。



这些函数的格式字符串都不支持浮点数。

13.3.7 字符设备驱动例程

在这里，我们将编写一个字符设备驱动的例程。这个字符设备与任何具体的硬件设备无关，它的功能是将应用程序写入的数据保存在一小块内存中，当应用程序读设备时也从这块内存中获取数据。

我们将这个例程命名为 `kpad`，源码中与设备相关的标识符的命名都加上了 `kpad` 作为前缀，这样做的好处是避免与内核定义的标识符重名。

13.3.7.1 源码

例程的源代码如下：

```
/* 文件名: kpad.c */
/* 说明: kpad 字符设备驱动例程 */
```



```
#include <linux/module.h> /* 模块编程 */
#include <linux/fs.h> /* 设备号、文件操作接口 */
#include <linux/cdev.h> /* 字符设备编程接口 */
#include <linux/uaccess.h> /* 访问用户态内存接口 */

MODULE_LICENSE("GPL"); /* 版权声明 */

#define BUF_SIZE 256 /* 内核缓冲区的大小 */

/* 以下类型表示 kpad 设备 */
struct kpad_dev {
    struct cdev cdev; /* 内嵌一个字符设备作为代表向内核注册 */
    char *buf; /* 保存内核缓冲区的首地址 */
    int len; /* 保存内核缓冲区的长度 */
    dev_t dev; /* 保存对应的设备号 */
};

/* 打开操作 */
static int kpad_open(struct inode *inode, struct file *file)
{
    /* 由 inode 对应的字符设备指针得到 kpad 设备的地址 */
    struct kpad_dev *dev = container_of(inode->i_cdev, struct kpad_dev, cdev);
    pr_debug("kpad_open: be called!\n");
    /* 将 kpad 设备的地址保存在打开的文件中 */
    file->private_data = dev;
    /* 返回 0 表示打开成功 */
    return 0;
}

/* 关闭操作 */
static int kpad_release(struct inode *inode, struct file *file)
{
    pr_debug("kpad_release: be called!\n");
    /* 无操作, 返回 0 表示成功 */
    return 0;
}

/* 读操作 */
static ssize_t kpad_read(struct file *file, char __user *buf,
                        size_t count, loff_t *pos)
{
    /* 从 file 中得到对应的 kpad 设备 */
    struct kpad_dev *dev = (struct kpad_dev *)file->private_data;
    pr_debug("kpad_read: pos = %d, count = %d.\n", (int)*pos, count);
    /* 参数合法性检查 */
    if (count < 0) return -EINVAL; /* 非法参数 */
    if (count == 0) return 0; /* 直接返回 0 */
    /* 控制读取的长度, 不能超过当前读写位置到内核缓冲区末尾的长度 */
    if (*pos+count > dev->len) count = dev->len-*pos;
    if (count == 0) return 0; /* 无数据可读, 返回 0 */
}
```



```

/* 从内核缓冲区复制数据到 buf 指向的用户态内存 */
if (copy_to_user(buf, dev->buf+*pos, count) > 0) {
    pr_debug("kpad_read: copy_to_user() FAILED!\n");
    return -EFAULT;
}
/* 修改文件的读写位置 */
*pos += count;
/* 返回读到的字节数 */
return count;
}

/* 写操作 */
static ssize_t kpad_write(struct file *file, const char __user *buf,
    size_t count, loff_t *pos)
{
    /* 从 file 中得到对应的 kpad 设备 */
    struct kpad_dev *dev = (struct kpad_dev *)file->private_data;
    pr_debug("kpad_write: pos = %d, count = %d.\n", (int)*pos, count);
    /* 参数合法性检查 */
    if (count < 0) return -EINVAL; /* 非法参数 */
    if (count == 0) return 0; /* 直接返回 0 */
    /* 控制写入的长度, 不能超过当前读写位置到内核缓冲区末尾的长度 */
    if (*pos+count > dev->len) count = dev->len-*pos;
    if (count == 0) return -ENOSPC; /* 无空间可写, 返回错误: 设备无空间 */
    /* 从 buf 指向的用户态内存复制数据到内核缓冲区 */
    if (copy_from_user(dev->buf+*pos, buf, count) > 0) {
        pr_debug("kpad_write: copy_from_user() FAILED!\n");
        return -EFAULT;
    }
    /* 修改文件的读写位置 */
    *pos += count;
    /* 返回写入的字节数 */
    return count;
}

/* 定位操作 */
static loff_t kpad_llseek(struct file *file, loff_t offset, int whence)
{
    /* 从 file 中得到对应的 kpad 设备 */
    struct kpad_dev *dev = (struct kpad_dev *)file->private_data;
    loff_t pos; /* 用于暂存新的读写位置 */
    pr_debug("kpad_llseek: f_pos = %d.\n", (int)file->f_pos);
    switch (whence) { /* 根据 whence 参数采用不同的操作 */
        case SEEK_SET: /* 相对于文件开始 */
            pos = offset;
            break;
        case SEEK_CUR: /* 相对于当前位置 */
            pos = file->f_pos+offset;
            break;
        case SEEK_END: /* 相对于文件结尾 */
            pos = dev->len+offset;

```

```

        break;
    default:
        return -EINVAL;
    }
    /* 新的读写位置应在合理范围之内 */
    if (pos < 0 || pos > dev->len) return -EINVAL;
    file->f_pos = pos; /* 修改文件的读写位置 */
    return pos; /* 返回新的读写位置 */
}

/* 设备的文件操作, 全局变量 */
static struct file_operations kpad_fops = {
    .owner = THIS_MODULE, /* 所属模块 */
    .open = kpad_open, /* 打开操作 */
    .release = kpad_release, /* 关闭操作 */
    .read = kpad_read, /* 读操作 */
    .write = kpad_write, /* 写操作 */
    .llseek = kpad_llseek, /* 定位操作 */
};

/* 全局变量, 表示一个 kpad 设备 */
static struct kpad_dev kpad;

/* 模块的初始化函数 */
static __init int kpad_init(void)
{
    int err; /* 用于保存错误码 */
    pr_debug("kpad_init: be called!\n");
    /* 分配设备号 */
    if ((err = alloc_chrdev_region(&kpad.dev, 0, 1, "kpad")) < 0) {
        pr_debug("kpad_init: alloc_chrdev_region() ERR = %d!\n", -err);
        goto alloc_chrdev_region_fail;
    }
    pr_debug("kpad_init: dev = (%d, %d).\n", MAJOR(kpad.dev), MINOR(kpad.dev));
    /* 分配内核缓冲区 */
    if ((kpad.buf = kmalloc(BUF_SIZE, GFP_KERNEL)) == NULL) {
        pr_debug("kpad_init: kmalloc() ERR!\n");
        err = -ENOMEM; /* 此错误号表示内存不足 */
        goto kmalloc_kpad_buf_fail;
    }
    /* 保存缓冲区的长度 */
    kpad.len = BUF_SIZE;
    /* 初始化字符设备 */
    cdev_init(&kpad.cdev, &kpad_fops);
    kpad.cdev.owner = kpad_fops.owner;
    /* 注册字符设备 */
    if ((err = cdev_add(&kpad.cdev, kpad.dev, 1)) < 0) {
        pr_debug("kpad_init: cdev_add() ERR = %d!\n", -err);
        goto cdev_add_fail;
    }
    return 0; /* 返回 0 表示初始化成功 */
}

```

```

/* 以下为出错处理，各个清理操作按初始化时的倒序排列，出错时跳转到相应的位置，
 * 这是驱动编程中常用的做法 */
cdev_add_fail:
    kfree(kpad.buf); /* 释放所分配的内存 */
kmallocc_kpad_buf_fail:
    unregister_chrdev_region(kpad.dev, 1); /* 注销所注册的设备号 */
alloc_chrdev_region_fail:
    return err;
}
module_init(kpad_init); /* 指定 kpad_init 为模块的初始化函数 */

/* 模块的退出函数 */
static __exit void kpad_exit(void)
{
    pr_debug("kpad_exit: be called!\n");
    /* 模块退出时进行清理，按初始化时的倒序操作 */
    cdev_del(&kpad.cdev); /* 注销字符设备 */
    kfree(kpad.buf); /* 释放所分配的内存 */
    unregister_chrdev_region(kpad.dev, 1); /* 注销所注册的设备号 */
}
module_exit(kpad_exit); /* 指定 kpad_exit 为模块的退出函数 */

```

13.3.7.2 说明

在设备的打开操作里，使用了如下的语句以获得指向 `kpad` 设备的指针：

```
struct kpad_dev *dev = container_of(inode->i_cdev, struct kpad_dev, cdev);
```

然后将这个指针保存在 `file->private_data` 中，在设备的其他操作里，又使用如下的语句以获得指向 `kpad` 设备的指针：

```
struct kpad_dev *dev = (struct kpad_dev *)file->private_data;
```

实际上，在上述的驱动源码中，代表 `kpad` 设备的变量被定义为全局变量 `kpad`，因此它的地址可以直接通过 `&kpad` 获得。在各种文件操作中之所以没有这样做，是为了让这些操作函数相对“独立”，尽可能少地与其他源码发生关联，以提高代码的可维护性。例如，如果要对上述驱动进行修改，以同时支持数个 `kpad` 设备，则这些操作函数都可以原封不动地用于各个设备。

`container_of` 是一个宏，它的第二和第三个参数分别是一个结构体类型的名称及它的某个成员的名称，第一个参数则要求是一个指针，指向这个类型的变量中的相应成员，返回的结果则是指向这个变量的指针。为了更好地理解，将它的定义写在下面：

```

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
#define container_of(ptr, type, member) ({ \
    const typeof( ((type *)0)->member ) *__mptr = (ptr); \
    (type *) ( (char *)__mptr - offsetof(type, member) ); })

```

这里 `offsetof` 宏通过取位于地址 0 处的结构体的成员地址而得到成员地址相对于结构体本身地址的偏移量，`container_of` 宏再通过成员地址减去这个偏移量得到结构体本身的地址。



13.3.7.3 测试

将这个例程编译、加载，用 `dmesg` 命令可从内核的调试信息中得到主次设备号，然后用 `mknod` 命令建立设备文件：

```
mknod -m 666 kpad c 251 0 # 假定主次设备号分别为 251 和 0
```

很多系统命令可以像操作普通文件那样操作设备文件，例如以下命令可将数据写入设备文件：

```
echo abcd > kpad # 将字符串 abcd 写入文件 kpad
```

`cat` 命令可以读设备文件，例如：

```
cat kpad # 读文件 kpad 的内容并显示在终端屏幕上
```

显然，这些命令在读写设备文件之前一定有打开操作，读写完毕之后有关闭操作。

Debian 5.0 的内核版本为 2.6.26，所提供的例程可以在这个版本的内核上正常编译和测试，同时也兼容 2.6.30 版本的内核。



第 14 章 内核编程接口

这一章主要对内核编程的常用接口进行介绍。内核编程的接口非常繁杂，并且随着内核版本的升高是不断变化的，因此本章尽量选择那些相对比较成熟、比较固定并且使用比较广泛的接口进行介绍，主要包括：

- ◆ 等待队列
- ◆ 定时器
- ◆ tasklet 和工作队列
- ◆ 自旋锁
- ◆ 互斥体与信号灯
- ◆ 端口 IO 与内存映射 IO
- ◆ 中断

通过学习一些常用的内核编程接口，一方面可以扩大驱动编程的自由度，另一方面可以了解一些内核的工作原理，提高驱动编程的代码质量。

14.1 双向环形链表

内核提供了关于双向环形链表数据结构的接口。这些接口不仅在内核源码中大量使用，而且可以在驱动编程时使用以减少代码量和出错的可能。要使用这些接口，必须包含以下头文件：

```
#include <linux/list.h>
```

14.1.1 定义与初始化

双向环形链表数据节点的定义如下：

```
struct list_head {  
    struct list_head *next; /* 指向后一个节点的指针 */  
    struct list_head *prev; /* 指向前一个节点的指针 */  
};
```

实际使用时，可以定义一个这种类型的变量作为链表头，如：

```
struct list_head my_list; /* 定义变量 my_list 作为链表头 */
```

所定义的链表头在使用前必须初始化，可以用下面的接口函数：

```
void INIT_LIST_HEAD(struct list_head *list);
```

参数 `list` 指向被初始化的链表头。初始化实际上就是让链表头的两个指针成员指向链表头自己，形成环形的结构。必须注意，在这种环形链表里，链表头本身不算链表的节点，因此初始化以后是一个空链表。

内核还提供了—个宏，用来定义链表头并同时将其初始化，使用方法如下：

```
LIST_HEAD(my_list); /* 定义变量 my_list 作为链表头，并且初始化 */
```

实际上，它就等价于以下代码：

```
struct list_head my_list = {&(my_list), &(my_list)};
```

也可以用内核定义的初始化符进行初始化，如：

```
struct list_head my_list = LIST_HEAD_INIT(my_list);
```

初始化符的定义如下：

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }
```

这是结构体变量初始化的语法，因此它只能用于初始化，不能用于给变量赋值。它的好处是可以用来给嵌套在结构体中的链表节点初始化，如：

```
struct my_struct {  
    struct list_head list;  
} my_node = { LIST_HEAD_INIT(name) };
```

在内核提供的很多编程接口中都有类似的成套定义。

14.1.2 链表操作

对链表进行操作的函数一般需要提供链表头或节点的地址作为参数，为使叙述简化，以下将直接用“节点 *x*”表示“参数 *x* 指向的节点”，用“链表 *y*”表示“参数 *y* 指向的链表”。

14.1.2.1 添加与删除节点

在链表头部添加新节点：

```
void list_add(struct list_head *new, struct list_head *head);
```

这个函数的功能是将节点 *new* 添加到链表 *head* 头部。

在链表尾部添加新节点：

```
void list_add_tail(struct list_head *new, struct list_head *head);
```

这个函数的功能是将节点 *new* 添加到链表 *head* 尾部。

从链表中删除节点：

```
void list_del(struct list_head *entry);
```

这个函数的功能是将节点 *entry* 从链表中删除。需要注意的是，删除前节点 *entry* 必须在某个链表中，删除后它的两个指针成员都指向非法内存。

从链表中删除节点，并将被删除的节点初始化：

```
void list_del_init(struct list_head *entry);
```


这个函数的功能是将节点 `entry` 从链表中删除，并将节点 `entry` 初始化。

14.1.2.2 替换与移动节点

用新节点替换链表中的旧节点：

```
void list_replace(struct list_head *old, struct list_head *new);
```

这个函数的功能是用节点 `new` 替换链表中的节点 `old`。

用新节点替换链表中的旧节点，并将旧节点初始化：

```
void list_replace_init(struct list_head *old, struct list_head *new);
```

这个函数的功能是用节点 `new` 替换链表中的节点 `old`，并将节点 `old` 重新初始化。

移动链表中的节点到另一个链表的头部：

```
void list_move(struct list_head *list, struct list_head *head);
```

这个函数的功能是将节点 `list` 从原来的链表移动到链表 `head` 头部。注意移动前节点 `list` 必须在某个链表中。

移动链表中的节点到另一个链表的尾部：

```
void list_move_tail(struct list_head *list, struct list_head *head);
```

这个函数的功能是将节点 `list` 从原来的链表移动到链表 `head` 尾部。注意移动前节点 `list` 必须在某个链表中。

14.1.2.3 链表连接与分割

连接两个链表：

```
void list_splice(const struct list_head *list, struct list_head *head);
```

这个函数的功能是将链表 `list` 连接到链表 `head` 头部。

连接两个链表：

```
void list_splice_tail(struct list_head *list, struct list_head *head);
```

这个函数的功能是将链表 `list` 连接到链表 `head` 尾部。

连接两个链表，并将剩下的链表头初始化：

```
void list_splice_init(struct list_head *list, struct list_head *head);
```

这个函数的功能是将链表 `list` 连接到链表 `head` 头部，这样 `list` 就成了一个孤立的链表头，然后将其初始化。

连接两个链表，并将剩下的链表头初始化：

```
void list_splice_tail_init(struct list_head *list, struct list_head *head);
```

这个函数的功能是将链表 `list` 连接到链表 `head` 尾部，这样 `list` 就成了一个孤立的链表头，然后将其初始化。

分割链表:

```
void list_cut_position(struct list_head *list,
                      struct list_head *head, struct list_head *entry);
```

这个函数的功能是将链表 `head` 一分为二, 在其节点 `entry` 之前的节点 (包括 `entry`) 被移动到链表 `list` 中, 剩下的节点仍留在链表 `head` 中。

注意, 节点 `entry` 必须在链表 `head` 中, 如果 `entry` 等于 `head`, 将不进行分割。如果分割前 `list` 不是空的, 则它的内容将丢失。

14.1.2.4 相关判断

判断是否为最后一个节点:

```
int list_is_last(const struct list_head *list, const struct list_head *head);
```

这个函数的功能是判断节点 `list` 是否为链表 `head` 的最后一个节点, 是则返回非 0, 否则返回 0。

判断链表是否为空:

```
int list_empty(const struct list_head *head);
```

这个函数的功能是判断链表 `head` 是否为空, 是则返回非 0, 否则返回 0。

判断链表是否只有一个节点:

```
int list_is_singular(const struct list_head *head);
```

这个函数的功能是判断链表 `head` 是否只有一个节点, 是则返回非 0, 否则返回 0。

14.1.3 链表的使用

内核提供的链表节点类型只包含两个指针成员, 不可能保存其他有用数据。一般的用法是将它嵌在需要组成链表的结构体类型中, 如:

```
struct my_data {
    struct list_head list; /* 用于组成链表 */
    int data; /* 其他有用数据 */
};
```

进行链表操作时, 以其 `list` 成员作为代表, 如:

```
struct my_data *new = kmalloc(sizeof(struct my_data), GFP_KERNEL);
list_add(&new->list, &my_list);
```

从链表头出发可以得到它的各个节点的指针, 显然, 这些指针指向 `struct my_data` 型数据的 `list` 成员, 因此可以用 `container_of` 宏得到指向 `struct my_data` 型数据的指针。在这里, 内核又将其重新定义为另一个宏:

```
#define list_entry(ptr, type, member) container_of(ptr, type, member)
```

因此可以用类似以下的代码访问所需的数据：

```
struct list_head *pos; /* 循环变量 */
for (pos = my_list.next; pos != &my_list; pos = pos->next) {
    struct my_data *p = list_entry(pos, struct my_data, list);
}
```

这个循环对链表中的节点进行了遍历。

一个有三个节点的双向环形链表的链接关系如图 14.1 所示，其中每个箭头代表一个指针的指向，链表节点左边的方框代表 **prev** 指针，右边的方框代表 **next** 指针。

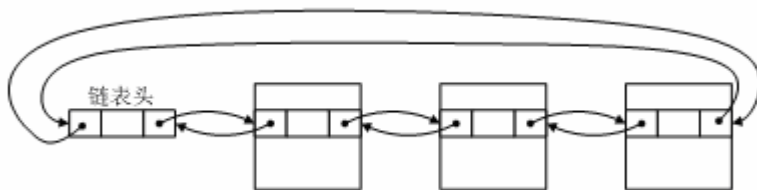


图 14.1 双向环形链表示意

内核还提供了一系列的宏以进行链表节点的遍历，这些宏实际上都是不带循环体的 **for** 语句，使用时直接在后面接循环体即可。其中最基本的一个宏形式如下：

```
list_for_each(pos, head)
```

这里参数 **pos** 是循环变量，必须在使用前定义，类型为 **(struct list_head *)**，参数 **head** 则指向要遍历的链表。每次循环，变量 **pos** 将指向链表中的下一个节点，遍历的顺序是从前往后。如果要从后往前遍历，则要用以下的形式：

```
list_for_each_prev(pos, head)
```

使用以上两个宏时，循环变量 **pos** 指向节点本身，多数情况下需要用 **list_entry** 转换成指向包含它的结构体的指针。为此，内核又提供了两个宏，可以在遍历的同时进行这种转换：

```
list_for_each_entry(pos, head, member) /* 从前往后遍历 */
list_for_each_entry_reverse(pos, head, member) /* 从后往前遍历 */
```

这里参数 **pos** 还是循环变量，但它不再用于指向链表节点，而是指向包含它的结构体类型，参数 **head** 仍然指向要遍历的链表，参数 **member** 则是链表节点在结构体中的成员名。用法举例如下：

```
struct my_data *pos; /* 循环变量 */
list_for_each_entry(pos, &my_list, list) { /* list 是 my_list 中的链表节点成员 */
    pos->data = 0; /* 访问循环变量指向的结构体 */
}
```

以上几个宏在使用时需要注意，在循环体内不能将循环变量指向的节点（当前节点）从链表中删除，因为这样它的指针就不再指向链表中的下一个节点，而循环中正是通过这个指针指到下一个节点的地址的。为了能在循环体内对当前节点进行删除操作，内核又提供了以上几个宏对应的“安

全”版本：

```
list_for_each_safe(pos, n, head)
list_for_each_prev_safe(pos, n, head)
list_for_each_entry_safe(pos, n, head, member)
list_for_each_entry_safe_reverse(pos, n, head, member)
```

与前面几个宏相比，它们多了一个参数 `n`，这是一个临时变量，用于暂存当前节点的指针值，这样即使当前节点被删除也可以正确地找到下一个节点。它必须在使用前定义，类型要与 `pos` 相同。

14.2 等待与延时

这一节主要介绍内核的等待队列编程接口，为此首先介绍进程调度与进程状态的概念，然后介绍系统的定时机制和延时操作。

14.2.1 调度与抢占

14.2.1.1 进程、线程与任务

在 Linux 应用编程中有进程和线程的概念。每个进程有自己独立的地址空间和独特的 PID（Process ID，进程标识符），进程的 PID 可通过 `getpid` 系统调用获得；而由进程创建的线程则与这个进程共享地址空间，如果是 NPTL（Native POSIX Thread Library）线程模型，则通过 `getpid` 系统调用得到的是进程的 PID。逻辑上，进程之间是独立的、相互隔离的，线程则附属于创建它的进程。

内核中是这样来实现进程与线程的。首先，不管是进程还是线程，都单独地执行流程，受内核统一的调度管理，这样的执行流程在内核中又称为任务。每个任务都有自己独特的 TID（Thread ID，线程标识符），并且属于某个线程组，由它的 TGID（Thread Group ID，线程组标识符）指明。一个线程组可以包含一个或多个任务，它们共享地址空间。线程组中有一个地位特殊的任务称为组长，它的 TID 等于 TGID。当应用程序创建一个新进程时，内核将创建一个新任务，新任务单独构成一个线程组并且是组长；当应用程序在进程中创建一个新线程时，内核创建的新任务将属于这个进程所在的线程组。

由于习惯的原因，内核编程中经常不加区别地使用进程、线程和任务三个概念。

应用程序通过 `getpid` 得到的实际上是 TGID。在一个线程中，可以通过以下接口函数得到线程自身的一个句柄：

```
#include <pthread.h>
pthread_t pthread_self(void);
```

注意它得到的并不是内核中的 TID，而是用户态内存中描述线程的一个结构体的首地址。如果要得到 TID，可使用以下系统调用：

```
pid_t gettid(void);
```

这个系统调用并非标准，而是 Linux 系统特有的。目前 glibc 库还不支持这个系统调用，但

可以由通用系统调用接口 `syscall` 来调用：

```
#include <unistd.h>
#include <sys/syscall.h> /* 系统调用编号的定义 */
int syscall(int number, ...);
```

其中 `number` 是系统调用的编号，一般会被定义为形如 `SYS_xxx` 的宏，例如：

```
pid_t tid = syscall(SYS_gettid); /* 获得线程的 TID */
```

注意如果不是 `NPTL` 线程模型，则同一进程内的各个线程通过 `getpid` 系统调用可能得到不同的值。

14.2.1.2 调度

Linux 是多任务的操作系统，可以同时执行多个进程。所谓“同时执行”应该这样来理解：如果是 **UP**（**Uniprocessor**，单处理器）的平台，微观上同一时刻只能执行一个进程，但由于 **CPU** 的执行流程快速地在多个进程间切换，宏观上给用户的感觉是这些进程在同时执行；如果是 **SMP**（**Symmetric Multiprocessor**，对称多处理器）的平台，每个 **CPU** 仍然有这样的行为，同时，还存在多个 **CPU** 真正同时执行多个进程的情况。这种使多个进程轮流占用 **CPU** 的操作称为进程调度。

内核中有一个非常核心的函数，用来实现这种进程间的切换，或者称为上下文切换。它的声明在头文件 `<linux/sched.h>` 中，原型如下：

```
void schedule(void);
```

这是一个既不需要参数又没有返回值的函数。在这个函数里，内核将决定哪一个进程将继续在当前 **CPU** 上执行。这个函数只能用在可以安全进行上下文切换的地方，一般用在进程上下文中。进程执行到这个函数以后，**CPU** 可能转而执行另外一个进程，当前进程因此失去执行的机会而停滞不前，直到有其他进程调用这个函数才有机会继续执行。当这个函数返回时，进程实际上已经在执行了。除了有调度的作用外，这个函数对进程的执行状态没有任何影响。

14.2.1.3 抢占

内核中可以发生上下文切换的地方称为调度点。如果调度点只存在于进程上下文中，也就是系统调用中，那么只有当进程调用了相关的系统调用才可能发生调度，这种系统称为非抢占式多任务的系统。在这种系统中，调度的时机完全可以由应用程序控制，如果进程不进行任何系统调用，那么它将一直占用 **CPU**。如果在某个中断内也加入调度点，并且这个中断是不断发生的，这种系统称为抢占式多任务的系统。在这种系统上，应用程序即使不主动调用引起进程调度的系统调用，也随时有可能发生进程调度。

Linux 是抢占式多任务的系统，它的调度点设置在所有陷阱、中断及部分异常返回用户态前。如果进程一直在内核态运行，则不会发生调度。为了解决这一问题，可以将内核配置并编译成抢占式内核，在这种配置下，当中断在内核态发生时也加入了调度点。由于抢占式内核仍被标记为“测试”性质的功能，大多数情况下并不使用。

一个进程正在执行时，在没有主动要求调度的情况下，执行流程被打断，**CPU** 去执行另外一个进程，这种行为称做后者对前者的抢占。抢占的实质是因为发生了中断，在中断内进行了调度。

14.2.2 进程运行状态

对应于每一个进程，内核中都有一个相应的结构体，用于保存进程相关的数据，它的类型定义如下：

```
struct task_struct {
    volatile long state; /* 进程的状态，0 为就绪或正在运行 */
    pid_t pid; /* 进程的全局 TID */
    pid_t tgid; /* 进程的全局 TGID */
    char comm[TASK_COMM_LEN]; /* 进程的可执行文件名，长度有限制 */
    /* 这里只写出了一小部分成员 */
};
```

这是一个庞大的结构体类型，上面只写出了其中的几个成员。

内核有一个数据结构，用于容纳所有处于就绪或正在运行状态的进程，称为执行队列。当调度发生时，内核将在执行队列中寻找一个最应该运行的进程，然后进行上下文切换，开始执行这个进程。

在上下文切换之前，内核将首先检查当前进程的状态，如果不是就绪或正在运行态，则将当前进程从执行队列中去除，这样它就不可能在以后的调度中得到运行的机会，在宏观上看来也停止了执行。进程离开执行队列可分为两种情况：一种情况是进程结束，永久离开执行队列；另一种情况是进程只是暂时离开执行队列，这时称它处于睡眠态。睡眠态的进程可以在将来重新回到执行队列。显然，进程不可能主动回到执行队列，必须由其他进程或中断修改它的状态并将其重新放回执行队列，这种操作称为唤醒。

进程的状态可由以下接口函数修改：

```
void set_task_state(struct task_struct *tsk, long state);
```

它实际上是一个宏，作用是修改参数 `tsk` 指向的进程的状态，其中参数 `state` 的常用值有以下几个。

- ◆ `TASK_RUNNING`：代表进程处于就绪或正在运行状态。
- ◆ `TASK_INTERRUPTIBLE`：代表进程处于睡眠态，但有信号发生时会被激活恢复运行。
- ◆ `TASK_UNINTERRUPTIBLE`：代表进程处于睡眠态，即使有信号发生进程也不会被激活。
- ◆ `TASK_KILLABLE`：代表进程处于睡眠态，有致命信号发生时会被激活恢复运行，对一般的信号无响应。致命信号是指那些没有被应用程序捕获的，默认操作是使应用程序退出或崩溃的信号。
- ◆ `TASK_STOPPED`：代表进程处于停止状态，通常是由 `STOP` 信号引起的，这种状态下也响应致命信号。
- ◆ `TASK_TRACED`：代表进程处于被跟踪的状态，通常是由于调试引起的，这种状态下也响应致命信号。

进行内核编程时常常需要知道当前正在执行的进程的相关数据，对此，内核提供了一个全局的指针 `current`，类型为 `(struct task_struct *)`。它的实现是平台相关的，但不管如何实现，内核保证它指向当前正在执行的进程对应的数据。如果是对称多处理器平台，多个 CPU 可以同时执行不

同的进程，因此各个 CPU 上得到的 `current` 指针也是不同的。如果要修改当前进程的状态，还可以使用以下接口函数：

```
void set_current_state(long state);
```

一般情况下，当前进程总是处于 `TASK_RUNNING` 状态，如果将它修改为其他状态，则在下一次调度时，进程将离开执行队列，进入睡眠态。

14.2.3 定时

因为 Linux 是抢占式多任务的系统，所以必须有一个中断不断地发生，并且不受应用程序的影响。系统的定时中断就满足这个条件，一般来说，它由定时器硬件支持，按固定的时间间隔发生，且时间间隔可由内核设置。在头文件 `<asm/param.h>` 中，内核定义了一个常数，表示每秒钟发生多少次这种中断：

```
#define HZ CONFIG_HZ
```

`CONFIG_HZ` 是内核的配置参数，可通过文件 `arch/arm/Kconfig` 修改。

内核还定义了一个全局变量 `jiffies`，类型是 `(unsigned long)`。在内核启动时，这个变量被初始化为 0，在系统的定时中断里，这个变量将增加 1，也就是说，它代表系统启动以来发生的定时中断的次数，相当于内核的一个计时装置。内核编程的很多接口函数都将 `jiffies` 作为一种时间单位来使用。

`jiffies` 变量的使用举例如下：

```
unsigned long t1, t2; /* 用来存放两个 jiffies 值 */
long diff, ms; /* 用来计算差值 */
t1 = jiffies; /* 记下开始时的 jiffies 值 */
do_some_thing();
t2 = jiffies; /* 记下结束时的 jiffies 值 */
diff = (long)t2 - (long)t1; /* 计算差值，转换成有符号型以避免时间回绕问题 */
ms = diff*1000/HZ; /* 转换为以毫秒为单位的时间 */
```

由于 `jiffies` 变量只是一个 32 位的整型，显然，在系统运行的某一时刻，它会由最大值变回 0，这样就发生了所谓“时间回绕”的问题。计算时可将 `jiffies` 值转换成有符号型以避免这一问题。如果只是对 `jiffies` 值进行比较，则可利用内核提供的以下接口：

```
time_after(a, b); /* 判断时间 a 是否在时间 b 之后 */
time_before(a, b); /* 判断时间 a 是否在时间 b 之前 */
time_after_eq(a, b); /* 判断时间 a 是否在时间 b 之后或等于 b */
time_before_eq(a, b); /* 判断时间 a 是否在时间 b 之前或等于 b */
time_in_range(a, b, c); /* 判断时间 a 是否在时间 b 和时间 c 之间（包含 b 和 c） */
```

这些都是宏定义，实际上采用的也是先将 `jiffies` 值转换为有符号型再比较的方式。

14.2.4 等待队列

需要让进程睡眠的原因是要等待某些条件的满足，比如，当进程读数据时，相应的设备还没有

数据，或者当进程写数据时，设备缓冲区已满，无法写入，这时可以让进程先进入睡眠状态，等设备可以进行读写操作时再将其唤醒。对此，内核提供了比较通用的机制，即等待队列。等待队列的相关定义与声明在以下头文件中：

```
#include <linux/wait.h>
```

14.2.4.1 定义和初始化

使用等待队列前，必须先定义一个变量，如：

```
wait_queue_head_t wq; /* 类型是 wait_queue_head_t */
```

实际上，这个数据类型内部包含了一个双向环形链表，用来存放正在等待的进程信息。在使用前必须将其初始化，可以使用以下接口函数：

```
void init_waitqueue_head(wait_queue_head_t *q);
```

还可以直接定义并且初始化，这要用到一个宏，例如：

```
DECLARE_WAIT_QUEUE_HEAD(wq); /* 定义等待队列 wq 并将其初始化 */
```

这条语句定义了一个等待队列，变量名为 **wq**，并且将其初始化。

等待队列也有一个初始化符，可以用它对变量进行初始化，例如：

```
wait_queue_head_t wq = __WAIT_QUEUE_HEAD_INITIALIZER(wq);
```

注意这个宏的名称前有两个下画线，一般来说，内核中名称以下画线开头的变量、函数、宏等表示它属于内部使用，对其作用没有充分了解的情况下尽量不要使用。

14.2.4.2 等待

让当前进程开始等待的最基本操作如下：

```
wait_event(wq, condition);
```

其中参数 **wq** 是进程要加入的等待队列，注意因为这是一个宏，所以这里不需要等待队列的指针就可以操作它。参数 **condition** 则是一个条件表达式。

这个宏实现的基本功能如下：开始时，如果 **condition** 为真，则当前进程仍为运行态，继续运行；如果 **condition** 为假，则当前进程进入睡眠态。当进程从睡眠态被唤醒时，如果 **condition** 为真，则进程恢复为运行态；如果 **condition** 为假，则进程再次进入睡眠态。简单地说，就是让进程进入等待状态，直到条件 **condition** 满足为止。

使用这个接口时，当前进程将进入不可被信号打断的睡眠态，也就是说不再响应信号。这种状态下的进程，即使向它发送 **KILL** 信号也不能将其中止，多数情况下这并不是想要的结果。为了让进程在等待时仍能响应信号，可使用以下接口：

```
wait_event_interruptible(wq, condition); /* 响应任何信号 */  
wait_event_killable(wq, condition); /* 只响应致命信号 */
```

这两个宏与 **wait_event** 功能相似，唯一不同的是当前进程将进入可被信号打断的睡眠状态，

当有信号发生时，进程将被激活继续执行。因此，它有一个返回值以区别被唤醒和被信号激活两种情况：返回 0 表示当前进程是通过等待队列被唤醒的，这种情况只有当条件表达式 `condition` 成立才有可能发生；返回错误码 `-ERESTARTSYS` 时表示当前进程是被信号激活的，这时条件 `condition` 不一定成立。

另外一个接口可以让当前进程等待一定的时间后自动恢复运行：

```
wait_event_timeout(wq, condition, timeout);
```

使用这个接口时，当前进程将进入不可被信号打断的睡眠态，但是当睡眠的时间超过参数 `timeout` 指定的时间间隔时，进程将自动醒过来，实际上是由一个内核定时器唤醒的。这个接口有返回值，表示进程醒来时离设定的超时还有多久。当返回值为 0 时，说明是因为超时而醒来的，这时条件 `condition` 不一定成立；当返回值非 0 时，说明是通过等待队列被唤醒的，这时条件 `condition` 一定是成立的。

还有一个接口综合了可被信号打断及有超时限制两个特征，形式如下：

```
wait_event_interruptible_timeout(wq, condition, timeout);
```

使用这个接口时，当前进程将进入可被信号打断的睡眠态，无论是发生信号还是超时，进程都会醒过来。它的返回值也是前两者的综合：返回正数或 0 表示进程醒来时离设定的超时还有多久，返回错误码 `-ERESTARTSYS` 时表示当前进程是被信号激活的。

进程还有一种特殊的开始等待的方式，称为排他式的等待。以这种方式入队将影响唤醒操作的行为，使用的接口如下：

```
wait_event_interruptible_exclusive(wq, condition);
```

它的参数与返回值的含义均与 `wait_event_interruptible` 宏相同。

14.2.4.3 唤醒

关于等待队列的另一方面的操作是唤醒，它的接口函数原型如下：

```
void wake_up(wait_queue_head_t *q);
```

它的功能是唤醒参数 `q` 指向的等待队列中所有以非排他方式入队的进程，对于以排他方式入队的进程则只唤醒第一个。

另外一个接口函数则可以将等待队列中的所有进程都唤醒，不管它是以排他方式还是非排他方式入队的，其原型如下：

```
void wake_up_all(wait_queue_head_t *q);
```

也可以只唤醒那些可以被信号打断的进程，使用以下接口函数：

```
void wake_up_interruptible(wait_queue_head_t *q);
```

这个函数仍然只能唤醒第一个以排他方式入队的进程。如果要唤醒全部可以被信号打断的进程，则使用以下接口函数：

```
void wake_up_interruptible_all(wait_queue_head_t *q);
```

实际上内核是这样来实现等待队列的：当进程以普通方式入队时放在队首，而以排他方式入队时放在队尾，并且在节点数据上设置一个排他标志。唤醒时的顺序是从队首到队尾，如果使用 `wake_up` 函数，则整个操作将在唤醒一个有排他标志的进程后终止；如果使用 `wake_up_all` 函数，则唤醒操作将持续到队尾。等待队列和唤醒的实现如图 14.2 所示。被唤醒的进程将离开所在的等待队列。

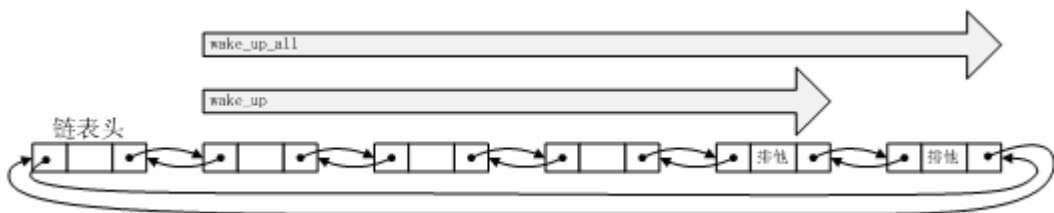


图 14.2 等待队列和唤醒



如果进程所等待的条件不成立，那么即使它被唤醒也会马上再进入睡眠状态。因此，一般都会先让进程所等待的条件成立之后再行唤醒操作。

唤醒操作可以用在中断上下文。

14.2.5 阻塞与非阻塞操作

进程进入睡眠状态，也称进程被阻塞或被挂起。应用程序对文件的读写操作有阻塞方式和非阻塞方式之分：阻塞方式下允许进程在读写操作中阻塞，非阻塞方式下则不允许。默认情况下，打开的文件是以阻塞方式操作的，如果应用程序要以非阻塞方式操作文件，则可以在打开文件时用如下代码设置标志位：

```
int fd;
fd = open("my_file", O_RDWR|O_NONBLOCK); /* 以可读写和非阻塞方式打开文件 */
```

这里 `O_NONBLOCK` 就是代表非阻塞方式的标志位，也可以在文件打开以后修改标志位，举例如下：

```
int flag = fcntl(fd, F_GETFL); /* 得到文件 fd 原来的标志 */
flag |= O_NONBLOCK; /* 增加 O_NONBLOCK 标志位 */
fcntl(fd, F_SETFL, flag); /* 将文件 fd 的标志设为新值 */
```

这些函数接口的声明都在应用编程的头文件 `<fcntl.h>` 中。

应用程序给文件设置的标志位保存在内核中对应的 `struct file` 型变量的 `f_flags` 成员中。驱动中可以通过如下方式检测非阻塞标志位：

```
if (file->f_flags & O_NONBLOCK) return -EAGAIN;
```

驱动在检测到文件有非阻塞的标志位后，就不应该阻塞进程的执行。对于文件的读写操作，一般应遵循如图 14.3 所示的流程。

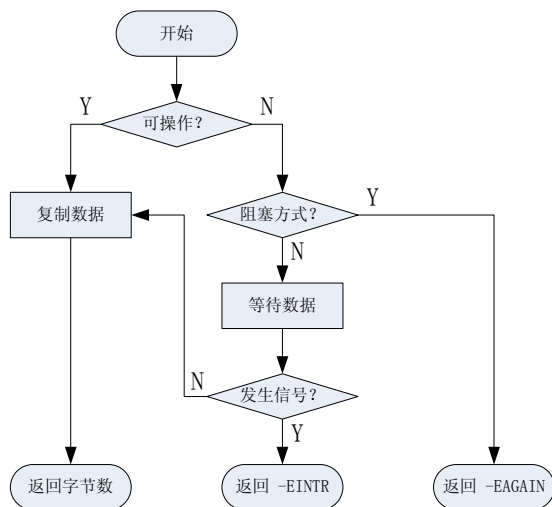


图 14.3 非阻塞方式的读写操作流程

14.2.6 延时

有时候需要进程的执行停滞一段时间再继续，称为延时。延时可分为两类：第一类为长延时，这种延时将使进程进入睡眠状态不再执行，将 CPU 让给其他进程；第二类为短延时，这种情况下进程实际上还在执行，要消耗 CPU 资源，只不过是在一个没有任何实际作用的空循环内。

短延时也可用在中断上下文内，使中断的执行过程延长。长延时只能用在进程上下文内。一般来说，长延时因为有进程的状态变化，时间尺度将远大于 CPU 的指令周期，而短延时的时间尺度可小到与 CPU 的指令周期相比拟。

由于短延时操作将持续消耗 CPU 资源，因此时间应尽量短，通常用在外部设备的速度跟不上 CPU 的执行速度时。

内核提供了延时的专用接口函数，声明在以下头文件中：

```
#include <linux/delay.h>
```

14.2.6.1 长延时

以毫秒为单位的长延时函数原型如下：

```
void msleep(unsigned int msecs);
```

其中参数 `msecs` 为要延时的毫秒数。

以秒为单位的长延时函数原型如下：

```
void ssleep(unsigned int seconds);
```

其中参数 `seconds` 为要延时的秒数。

以上两个函数将使进程进入不可被信号打断的睡眠态，如果要求可以响应信号，则要使用下面这个函数：

```
unsigned long msleep_interruptible(unsigned int msecs);
```

其参数及返回值的含义解释如下。

- ◆ msec: 要延时的毫秒数。
- ◆ 返回值: 剩余的毫秒数, 0 表示延时的时间到, 非 0 表示被信号打断。

14.2.6.2 短延时

以纳秒为单位的短延时函数原型如下:

```
void ndelay(unsigned long nsecs);
```

其中参数 `nsecs` 表示要延时的纳秒数。当然, 一般来说, 系统并不支持小到几个纳秒的延时。即使 `nsecs` 参数等于 1, 延时可能也在微秒量级。

以微秒为单位的短延时函数原型如下:

```
void udelay(unsigned long usecs);
```

其中参数 `usecs` 表示要延时的微秒数。

以毫秒为单位的短延时函数原型如下:

```
void mdelay(unsigned long msecs);
```

其中参数 `msecs` 表示要延时的毫秒数。

14.2.7 等待队列例程

在这个例程中, 我们将用等待队列实现设备的阻塞与非阻塞操作。设备的功能类似于一个 FIFO, 写入的数据保存在内核缓冲区中, 读取的时候按照先进先出的原则。被读过的数据从内核缓冲区中清除。如果缓冲区空, 则阻止进行读操作; 如果缓冲区满, 则阻止进行写操作。

14.2.7.1 源码

例程源代码如下:

```
/* 文件名: kpipe.c */
/* 说明: 等待队列例程 */

#include <linux/module.h> /* 模块编程 */
#include <linux/fs.h> /* 设备号、文件操作接口 */
#include <linux/cdev.h> /* 字符设备编程接口 */
#include <linux/uaccess.h> /* 访问用户态内存接口 */
#include <linux/sched.h> /* 进程调度相关 */
#include <linux/wait.h> /* 等待队列接口 */

MODULE_LICENSE("GPL"); /* 版权声明 */

#define BUF_SIZE 256 /* 内核缓冲区的大小 */

/* 以下类型表示 kpipe 设备 */
struct kpipe_dev {
```

```

struct cdev cdev; /* 内嵌一个字符设备作为代表向内核注册 */
char *buf; /* 保存内核缓冲区的首地址 */
int len; /* 保存内核缓冲区的长度 */
int head; /* 环形缓冲区头指针 */
int tail; /* 环形缓冲区尾指针 */
dev_t dev; /* 保存对应的设备号 */
wait_queue_head_t rq; /* 读等待队列 */
wait_queue_head_t wq; /* 写等待队列 */
struct mutex lock; /* 同步变量 */
};

/* 判断设备缓冲区是否空 */
static inline int is_empty(const struct kpipe_dev *dev)
{
    return dev->head == dev->tail;
}

/* 判断设备缓冲区是否满 */
static inline int is_full(const struct kpipe_dev *dev)
{
    /* 让缓冲区的实际容量是 BUF_SIZE-1, 以避免无法区分空与满的情况,
     * 当尾指针正好在头指针之前时就认为缓冲区满 */
    return dev->tail+1 == dev->head ||
           (dev->tail == BUF_SIZE-1 && dev->head == 0);
}

/* 判断设备缓冲区中数据是否回绕 */
static inline int is_rewind(const struct kpipe_dev *dev)
{
    return dev->head > dev->tail;
}

/* 得到设备缓冲区中的数据个数 */
static inline int nr_bytes(const struct kpipe_dev *dev)
{
    if (is_rewind(dev)) {
        return dev->len - dev->head + dev->tail;
    } else {
        return dev->tail - dev->head;
    }
}

/* 得到设备缓冲区中的空位个数 */
static inline int nr_holes(const struct kpipe_dev *dev)
{
    if (is_rewind(dev)) {
        return dev->head - dev->tail - 1;
    } else {
        return dev->len - dev->tail + dev->head - 1;
    }
}

```



```

/* 打开操作 */
static int kpipe_open(struct inode *inode, struct file *file)
{
    /* 由 inode 对应的字符设备指针得到 kpipe 设备的地址 */
    struct kpipe_dev *dev = container_of(inode->i_cdev,
        struct kpipe_dev, cdev);
    pr_debug("kpipe_open: be called!\n");
    /* 将 kpipe 设备的地址保存在打开的文件中 */
    file->private_data = dev;
    /* 返回 0 表示打开成功 */
    return 0;
}

/* 关闭操作 */
static int kpipe_release(struct inode *inode, struct file *file)
{
    pr_debug("kpipe_release: be called!\n");
    /* 无操作, 返回 0 表示成功 */
    return 0;
}

/* 读操作 */
static ssize_t kpipe_read(struct file *file, char __user *buf,
    size_t count, loff_t *pos)
{
    int err; /* 用于暂存错误码 */
    int bytes; /* 用于保存读取的字节数 */
    /* 从 file 中得到对应的 kpipe 设备 */
    struct kpipe_dev *dev = (struct kpipe_dev *)file->private_data;
    pr_debug("kpipe_read: count = %d.\n", count);
    /* 参数合法性检查 */
    if (count < 0) return -EINVAL; /* 非法参数 */
    if (count == 0) return 0; /* 直接返回 0 */
    if (is_empty(dev)) { /* 缓冲区为空 */
        /* 非阻塞方式下返回 -EAGAIN */
        if (file->f_flags & O_NONBLOCK) return -EAGAIN;
        /* 等待直到缓冲区不空 */
        err = wait_event_interruptible(dev->rq, !is_empty(dev));
        /* 发生信号则返回 -EINTR */
        if (err == -ERESTARTSYS) return -EINTR;
    }
    /* 以下进入临界区, 不允许多个进程同时修改头尾指针 */
    mutex_lock(&dev->lock); /* 加锁 */
    /* 限制读取的个数不超过缓冲区中数据的个数 */
    if (count > nr_bytes(dev)) count = nr_bytes(dev);
    bytes = 0; /* 已读取的字节数为 0 */
    if (is_rewind(dev) && count > dev->len-dev->head) {
        /* 读取的区域经过缓冲区末尾, 需要读两次 */
        bytes = dev->len-dev->head; /* 计算要读取的字节数 */
        if (copy_to_user(buf, dev->buf+dev->head, bytes) > 0) {

```



```

        pr_debug("kpipe_read: copy_to_user() FAILED!\n");
        return -EFAULT;
    }
    dev->head = 0; /* 重设头指针使后续的读取从头开始 */
}
/* 如果是第二次读, 目标地址应后移 bytes, 读取个数减少 bytes */
if (copy_to_user(buf+bytes, dev->buf+dev->head, count-bytes) > 0) {
    pr_debug("kpipe_read: copy_to_user() FAILED!\n");
    return -EFAULT;
}
/* 更新头指针位置 */
dev->head += count-bytes;
mutex_unlock(&dev->lock); /* 解锁 */
/* 唤醒写操作阻塞的进程 */
wake_up_interruptible(&dev->wq);
return count;
}

/* 写操作 */
static ssize_t kpipe_write(struct file *file, const char __user *buf,
                          size_t count, loff_t *pos)
{
    int err; /* 用于暂存错误码 */
    int bytes; /* 用于保存写入的字节数 */
    /* 从 file 中得到对应的 kpipe 设备 */
    struct kpipe_dev *dev = (struct kpipe_dev *)file->private_data;
    pr_debug("kpipe_write: count = %d.\n", count);
    /* 参数合法性检查 */
    if (count < 0) return -EINVAL; /* 非法参数 */
    if (count == 0) return 0; /* 直接返回 0 */
    if (is_full(dev)) {
        /* 非阻塞方式下返回 -EAGAIN */
        if (file->f_flags & O_NONBLOCK) return -EAGAIN;
        /* 等待直到缓冲区不满 */
        err = wait_event_interruptible(dev->wq, !is_full(dev));
        /* 发生信号则返回 -EINTR */
        if (err == -ERESTARTSYS) return -EINTR;
    }
    /* 以下进入临界区, 不允许多个进程同时修改头尾指针 */
    mutex_lock(&dev->lock); /* 加锁 */
    /* 限制写入的个数不超过缓冲区中空位的个数 */
    if (count > nr_holes(dev)) count = nr_holes(dev);
    bytes = 0; /* 已写入的字节数为 0 */
    if (!is_rewind(dev) && count > dev->len-dev->tail) {
        /* 写入的区域经过缓冲区末尾, 需要写两次 */
        bytes = dev->len-dev->tail; /* 计算要写入的字节数 */
        if (copy_from_user(dev->buf+dev->tail, buf, bytes) > 0) {
            pr_debug("kpipe_read: copy_from_user() FAILED!\n");
            return -EFAULT;
        }
    }
    dev->tail = 0; /* 重设尾指针使后续的写入从头开始 */
}

```

```

    }
    /* 如果是第二次写, 目标地址应后移 bytes, 写入个数减少 bytes */
    if (copy_from_user(dev->buf+dev->tail, buf+bytes, count-bytes) > 0) {
        pr_debug("kpipe_read: copy_from_user() FAILED!\n");
        return -EFAULT;
    }
    /* 更新尾指针位置 */
    dev->tail += count-bytes;
    mutex_unlock(&dev->lock); /* 解锁 */
    /* 唤醒读操作阻塞的进程 */
    wake_up_interruptible(&dev->rq);
    return count;
}

/* 定位操作 */
static loff_t kpipe_llseek(struct file *file, loff_t offset, int whence)
{
    /* 因为 kpipe 设备类似于 FIFO, 没有读写位置的概念, 故直接返回错误码 */
    return -ESPIPE;
}

/* 设备的文件操作, 全局变量 */
static struct file_operations kpipe_fops = {
    .owner = THIS_MODULE, /* 所属模块 */
    .open = kpipe_open, /* 打开操作 */
    .release = kpipe_release, /* 关闭操作 */
    .read = kpipe_read, /* 读操作 */
    .write = kpipe_write, /* 写操作 */
    .llseek = kpipe_llseek, /* 定位操作 */
};

/* 全局变量, 表示一个 kpipe 设备 */
static struct kpipe_dev kpipe;

/* 模块的初始化函数 */
static __init int kpipe_init(void)
{
    int err; /* 用于保存错误码 */
    pr_debug("kpipe_init: be called!\n");
    /* 初始化互斥体 */
    mutex_init(&kpipe.lock);
    /* 分配设备号 */
    if ((err = alloc_chrdev_region(&kpipe.dev, 0, 1, "kpipe")) < 0) {
        pr_debug("kpipe_init: alloc_chrdev_region() ERR = %d!\n", -err);
        goto alloc_chrdev_region_fail;
    }
    pr_debug("kpipe_init: dev = (%d, %d).\n",
        MAJOR(kpipe.dev), MINOR(kpipe.dev));
    /* 分配内核缓冲区 */
    if ((kpipe.buf = kmalloc(BUF_SIZE, GFP_KERNEL)) == NULL) {
        pr_debug("kpipe_init: kmalloc() ERR!\n");
    }
}

```



```

    err = -ENOMEM; /* 此错误号表示内存不足 */
    goto kmalloc_kpipe_buf_fail;
}
/* 保存缓冲区的长度 */
kpipe.len = BUF_SIZE;
/* 初始化缓冲区头尾指针 */
kpipe.head = 0;
kpipe.tail = 0;
/* 初始化等待队列 */
init_waitqueue_head(&kpipe.rq);
init_waitqueue_head(&kpipe.wq);
/* 初始化字符设备 */
cdev_init(&kpipe.cdev, &kpipe_fops);
kpipe.cdev.owner = kpipe_fops.owner;
/* 注册字符设备 */
if ((err = cdev_add(&kpipe.cdev, kpipe.dev, 1)) < 0) {
    pr_debug("kpipe_init: cdev_add() ERR = %d!\n", -err);
    goto cdev_add_fail;
}
return 0; /* 返回 0 表示初始化成功 */
/* 以下为出错处理，各个清理操作按初始化时的倒序排列，出错时跳转到相应的位置，
 * 这是驱动编程中常用的做法 */
cdev_add_fail:
    kfree(kpipe.buf); /* 释放所分配的内存 */
kmalloc_kpipe_buf_fail:
    unregister_chrdev_region(kpipe.dev, 1); /* 注销所注册的设备号 */
alloc_chrdev_region_fail:
    mutex_destroy(&kpipe.lock); /* 销毁互斥体 */
    return err;
}
module_init(kpipe_init); /* 指定 kpipe_init 为模块的初始化函数 */

/* 模块的退出函数 */
static __exit void kpipe_exit(void)
{
    pr_debug("kpipe_exit: be called!\n");
    /* 模块退出时进行清理，按初始化时的倒序操作 */
    cdev_del(&kpipe.cdev); /* 注销字符设备 */
    kfree(kpipe.buf); /* 释放所分配的内存 */
    unregister_chrdev_region(kpipe.dev, 1); /* 注销所注册的设备号 */
    mutex_destroy(&kpipe.lock); /* 销毁互斥体 */
}
module_exit(kpipe_exit); /* 指定 kpipe_exit 为模块的退出函数 */

```

14.2.7.2 说明

在这个例程里，内核缓冲区使用的是一种环形缓冲区。为了实现环形缓冲区，必须有两个变量记住缓冲区中数据的开始和结束位置，称为头指针和尾指针。进行读操作时，数据将从头指针指示的位置开始读取，同时头指针后移；进行写操作时，数据从尾指针指示的位置写入，同时尾指针后移。如果指针移动到缓冲区最后面，则让它返回缓冲区开始处。

当缓冲区中无数据时，头指针与尾指针相等。当缓冲区中全部写满数据时，头指针与尾指针也相等。为了区分空与满两种情况，可以采用以下两种方案。

- ◆ 第一种方案需要一个额外的标志变量，它有两个取值，表示缓冲区的“空”与“满”。当读操作导致头尾指针相等时，置标志为“空”；当写操作导致头尾指针相等时，置标志为“满”。检测时要同时检测头尾指针是否相等以及这个标志的值。
- ◆ 第二种方案不需要额外的变量，但要牺牲缓冲区中的一个存储空间，也就是说，当尾指针正好在头指针的前一个位置时，就认为缓冲区已满，这样就人为地把空与满的情形区分开了。本例程中采用的就是这种方案。

写入数据时，如果到达缓冲区的末尾，则尾指针将返回缓冲区开始处继续写入，这样数据就被分成了两段，我们称之为“数据回绕”。遇到这种情况，复制操作必须分段处理。

由于读写操作只允许发生在指定的位置，所以对应用程序来说，文件的读写位置已经失去了意义。关于同步变量的使用将在后续的自旋锁与同步一节中说明。

本例程在 Linux 2.6.26 和 Linux 2.6.30 内核上编译并测试通过。

14.3 定时器与延期工作

定时器是内核提供了一种编程接口，使用它可以在将来的一个指定时刻执行一段指定的代码。而延期工作指的是要执行的代码不在当前上下文内立刻执行，而是注册到内核，由内核在将来的某个时刻执行。定时器与延期工作的机制有一定的相似性，因此把它们放在同一节内介绍。

14.3.1 定时器

使用定时器可以让内核在将来的一个指定时刻执行指定的代码，其相关定义与声明在以下头文件中：

```
#include <linux/timer.h>
```

14.3.1.1 定义与初始化

定时器与一个结构体类型有关，其定义如下：

```
struct timer_list {
    struct list_head entry; /* 链表节点 */
    unsigned long expires; /* 过期时间 */
    void (*function)(unsigned long); /* 工作函数 */
    unsigned long data; /* 工作函数参数 */
    struct tvec_base *base; /* 内核内部使用 */
};
```

其中部分成员的含义解释如下。

- ◆ entry: 链表节点，可以让定时器变量与内核的定时器列表联系起来。
- ◆ expires: 过期时间，jiffies 值，当系统 jiffies 值超过这个值时，定时器的函数就要

被执行。

- ◆ **function**: 定时器的工作函数，这是一个函数指针。
- ◆ **data**: 工作函数参数，调用工作函数时使用。

要使用定时器，必须先用这个类型定义一个变量，如：

```
struct timer_list my_timer; /* 定义定时器变量 my_timer */
```

变量在使用前必须初始化，可以使用下面这个函数：

```
void init_timer(struct timer_list *timer);
```

其中 **timer** 参数指向被初始化的定时器变量。但是这个函数只执行基本的初始化操作，因此常用的是另外一个函数，其完整定义如下：

```
static inline void setup_timer(struct timer_list * timer,
                              void (*function)(unsigned long), unsigned long data)
{
    timer->function = function;
    timer->data = data;
    init_timer(timer);
}
```

可见使用这个函数时，会将工作函数和工作函数参数也同时设置好。现在只剩过期时间没有设置，必须直接赋值，如：

```
my_timer.expires = jiffies + HZ; /* 过期时间设为当前时间加 HZ，即 1 秒之后 */
```

定时器也可以直接定义并初始化，如：

```
DEFINE_TIMER(name, function, expires, data); /* 定义定时器，变量名为 name */
```

其各个参数的含义解释如下。

- ◆ **name**: 定时器变量的名称。
- ◆ **function**: 工作函数。
- ◆ **expires**: 过期时间。
- ◆ **data**: 工作函数参数。

也可以使用初始化符，例如：

```
struct timer_list my_timer = TIMER_INITIALIZER(function, expires, data);
```

注意这种初始化时给的过期时间一般没有什么意义，通常在注册定时器时才设置过期时间。

14.3.1.2 注册与注销

要使定时器起作用，必须将其注册，注册定时器的接口函数原型如下：

```
void add_timer(struct timer_list *timer);
```

其中参数 `timer` 指向被注册的定时器变量。注册以后，定时器变量就被放到内核的定时器列表中，当时间流逝到它的过期时间之后，内核就会执行它的工作函数。



如果设置的过期时间已经比当前时间早，工作函数也会被执行一次。

当内核准备执行一个定时器的工作函数（简称为执行这个定时器）时，首先要把定时器变量从定时器列表中删除，因此定时器注册一次只能执行一次。这里把定时器注册以后到执行之前的状态称为激活状态。实际上，定时器是否处在激活状态与它是否在内核的定时器列表里相关。

注销定时器的接口函数原型如下：

```
int del_timer(struct timer_list *timer);
```

其参数及返回值的含义解释如下。

- ◆ `timer`：指向被注销的定时器变量。
- ◆ 返回值：0 表示注销了一个非激活的定时器，1 表示注销了一个激活的定时器。

注销操作将使定时器变量离开内核的定时器列表，变为非激活状态。如果将它用于非激活的定时器，则等于无操作。

在对称多处理器的平台上，当进行注销操作时，定时器有可能正在另外一个 CPU 上执行。为了确保注销以后定时器不再执行，应使用以下的同步注销接口函数：

```
int del_timer_sync(struct timer_list *timer);
```

这个函数内部使用了同步机制，当定时器正在执行时，它会一直等待到定时器执行完毕再返回，这样就确保了函数返回后定时器一定不再执行。这种等待类似于短延时，是一种消耗 CPU 的“忙等待”。在 UP 平台上，它与 `del_timer` 是相同的。

14.3.1.3 修改

当定时器处于激活状态时，因为它在内核的定时器列表中，故不能随意修改。如果要修改定时器的过期时间，可使用以下接口函数：

```
int mod_timer(struct timer_list *timer, unsigned long expires);
```

其各个参数及返回值的含义解释如下。

- ◆ `timer`：指向被修改的定时器变量。
- ◆ `expires`：新的过期时间。
- ◆ 返回值：0 表示修改了一个非激活的定时器，1 表示修改了一个激活的定时器。

它的功能可以简单地理解如下：

```
del_timer(timer); /* 注销定时器 */
timer->expires = expires; /* 修改定时器的过期时间为新时间 */
add_timer(timer); /* 注册定时器 */
```

即定时器被修改完之后还要被注册。因此实际上它可以代替 `add_timer`，作为一个可以同时设置过期时间并注册定时器的函数来使用。

还有一个函数只修改激活状态的定时器的过期时间，对于非激活状态的定时器则不做处理，原型如下：

```
int mod_timer_pending(struct timer_list *timer, unsigned long expires);
```

其各个参数的含义与 `mod_timer` 函数相同，返回值为 0 表示定时器处于非激活状态，未修改，1 表示修改成功。

14.3.1.4 判断状态

内核提供了一个判断定时器是否在激活状态的函数，定义如下：

```
static inline int timer_pending(const struct timer_list * timer)
{
    return timer->entry.next != NULL;
}
```

其中参数 `timer` 指向要判断的定时器，返回值是 0 表示定时器不在激活状态，非 0 表示定时器在激活状态。实际上它就是通过判断定时器是否在一个链表中而得出结论的。

14.3.1.5 使用

定时器并不在进程上下文内执行。当定时器在执行时，注册它的进程完全有可能在睡眠状态甚至已退出。实际上，定时器的工作函数在内核的软中断中执行，因此属于中断上下文。

定时器的相关函数接口都可以在中断上下文中使用，利用这一点可以让内核周期性地执行某个函数，方法是在定时器的工作函数内再注册自身。

初始化定时器时可以向定时器的工作函数传递一个参数，虽然它只是一个长整型，但是可以利用强制转换的方法传递一个指针值。这样，只要将数据包装成一个结构体，就可以全部传递到工作函数中去。



不要将局部变量的指针传递到定时器的工作函数中去，因为当定时器执行时，局部变量很可能已经消失，从而导致非法内存访问。

示例代码如下：

```
static struct timer_data {
    struct timer_list timer; /* 将定时器作为成员 */
} timer_data; /* 定义变量，包含一个定时器成员 */

/* 定时器的工作函数 */
static void timer_fn(unsigned long d)
{
    struct timer_data *p = (struct timer_data *)d;
    /* 必须有一个条件控制是否需要再注册自身，否则定时器将永不停止 */
    if (/* 需要再注册自身 */) mod_timer(&p->timer, p->timer.expires+HZ);
}
```



```

}

/* 在某处注册定时器 */
void fun(void)
{
    setup_timer(&timer_data.timer, timer_fn, (unsigned long)&timer_data);
    timer_data.timer.expires = jiffies+HZ;
    add_timer(&timer_data.timer);
}

```

定时器还有一个特性是它的工作函数一定与注册它的那段代码在同一个 CPU 上执行。

14.3.2 tasklet

tasklet 接口可以用来延期执行一个函数，相关的定义和声明在以下头文件中：

```
#include <linux/interrupt.h>
```

14.3.2.1 定义与初始化

tasklet 与一个结构体类型有关，其定义如下：

```

struct tasklet_struct
{
    struct tasklet_struct *next; /* 链表的 next 指针 */
    unsigned long state; /* 状态位 */
    atomic_t count; /* 使能/禁用标志，内核内部使用 */
    void (*func)(unsigned long); /* 工作函数 */
    unsigned long data; /* 工作函数参数 */
};

```

可见，**tasklet** 也有与定时器相同的工作函数和工作函数参数。两者的功能也有类似的地方，都是让一个函数在将来执行。所不同的是定时器有过期时间的概念，它的工作函数在系统时间流逝到这个时间之后才执行；而 **tasklet** 的工作函数则是“尽快”执行，没有时间概念。两者都是由内核的软中断实现的，它们的工作函数都在中断上下文中执行。

使用 **tasklet** 时，首先定义一个变量，如：

```
struct tasklet_struct my_tasklet;
```

变量在使用前必须初始化，可以用以下函数：

```

void tasklet_init(struct tasklet_struct *t,
                 void (*func)(unsigned long), unsigned long data);

```

其各个参数的含义解释如下。

- ◆ **t**：指向被初始化的 **tasklet** 变量。
- ◆ **func**：函数指针，指向工作函数。
- ◆ **data**：传给工作函数的参数。



也可以用一个宏直接定义并初始化，例如：

```
DECLARE_TASKLET(my_tasklet, func, data);
```

其中 `my_tasklet` 是所定义的变量的名称，`func` 是工作函数指针，`data` 是传给工作函数的参数。

以上两种方法初始化后的 `tasklet` 变量都处于使能状态。内核提供了另外一个宏，也可以直接定义并初始化 `tasklet`，但初始状态为禁用状态，用法如下：

```
DECLARE_TASKLET_DISABLED(my_tasklet, func, data);
```

其各个参数的含义与 `DECLARE_TASKLET` 完全一致。

14.3.2.2 注册

注册就是将 `tasklet` 变量添加到内核的一个链表中。内核实际上定义了两个 `tasklet` 链表，一个为普通优先级，一个为高优先级。当软中断发生时，内核会先依次执行高优先级链表中 `tasklet` 变量的工作函数（也可简称为执行这个 `tasklet`），然后再依次执行普通优先级链表中的 `tasklet`，被执行的 `tasklet` 将从链表中删除。

注册 `tasklet` 的接口函数原型如下：

```
void tasklet_schedule(struct tasklet_struct *t); /* 注册为普通优先级 */
void tasklet_hi_schedule(struct tasklet_struct *t); /* 注册为高优先级 */
```

其中参数 `t` 指向要注册的 `tasklet` 变量。

与定时器相似，`tasklet` 也能保证它的工作函数与注册它的那段代码在同一个 CPU 上执行。`tasklet` 的实现上有同步机制，可以保证同一个 `tasklet` 不会同时在多个 CPU 上执行。

与定时器不同的是，注册 `tasklet` 时，内核会先检查它是否已注册（状态保存在其 `state` 成员中），如果未注册才将其加入链表中，如果已注册则什么也不做。所以，一个 `tasklet` 在被执行前可以安全地注册多次，但只会被执行一次。

内核并没有提供简单的方法以撤销一个已注册的 `tasklet` 的执行，但是提供了一个接口函数，可以确保某个 `tasklet` 不再执行，原型如下：

```
void tasklet_kill(struct tasklet_struct *t);
```

其中参数 `t` 指向要操作的 `tasklet` 变量。`tasklet_kill` 函数会不断地检测参数 `t` 指向的 `tasklet` 变量，如果它在注册状态，则让出 CPU 使之有执行机会。由于整个过程中有进程调度，所以这个函数不能用在中断上下文中。



如果 `tasklet` 在其工作函数里注册自身的话，则 `tasklet_kill` 调用返回后仍不能保证它在未注册状态，这时必须用某种方法在调用 `tasklet_kill` 之前阻止 `tasklet` 注册自身。

14.3.2.3 禁用与使能

`tasklet` 可以被禁用，禁用以后即使它处于内核的 `tasklet` 链表中也不会被执行，其接口函数

原型如下：

```
void tasklet_disable(struct tasklet_struct *t);
```

其中参数 `t` 指向要被禁用的 `tasklet`。在 SMP 平台上用这个函数做禁用操作时，如果 `tasklet` 已经在另外一个 CPU 上执行，那么会一直“忙等待”到它执行完成。这样，当函数调用返回后，可以确保 `tasklet` 已不在任何 CPU 上执行。如果不需要这种同步操作，则可以用下面这个函数：

```
void tasklet_disable_nosync(struct tasklet_struct *t);
```

其中 `t` 指向要被禁用的 `tasklet`。

被禁用的 `tasklet` 还可以被使能，其接口函数原型如下：

```
void tasklet_enable(struct tasklet_struct *t);
```

`tasklet` 变量包含一个成员 `count`，禁用时这个成员的值加 1，使能时这个成员的值减 1，它的值为 0 时相应的 `tasklet` 才会被执行。因此，如果一个 `tasklet` 初始状态为使能状态，当它被禁用两次以后，必须相应地使能两次才真正有效。这个成员被定义为原子型，以实现多个 CPU 之间的同步。

14.3.3 工作队列

无论是定时器还是 `tasklet`，它们的工作函数都执行在软中断上下文中，因此不允许发生进程调度，比如等待队列的等待操作等。如果要延期的工作不能放在中断上下文中，则可以考虑使用工作队列。工作队列的相关定义与声明放在以下头文件中：

```
#include <linux/workqueue.h>
```

14.3.3.1 工作的定义与初始化

工作由一个结构体类型描述，定义如下：

```
typedef void (*work_func_t)(struct work_struct *work); /* 工作函数类型 */

struct work_struct {
    atomic_long_t data; /* 状态位及工作队列指针，内核内部使用 */
    struct list_head entry; /* 链表节点 */
    work_func_t func; /* 工作函数 */
};
```

它的工作函数原型与定时器和 `tasklet` 有差别，传入参数不是无符号长整型而是指向结构体的指针，实际上，当它的工作函数被调用时，传入的指针就指向自己。如果希望向工作函数传递其他数据，可把 `work_struct` 结构体嵌套在另一个结构体类型中使用，如：

```
struct my_work_struct {
    struct work_struct work; /* 嵌套的 work_struct */
    int other_data; /* 其他数据 */
};
```


在工作函数中，用 `container_of` 宏得到指向外围结构体的指针，如：

```
void my_work_fn(struct work_struct *work)
{
    struct my_work_struct *p = container_of(work, struct my_work_struct, work);
    p->other_data = 0; /* 操作其他数据 */
}
```

工作在使用前必须初始化，接口函数原型如下：

```
void INIT_WORK(struct work_struct *work, work_func_t func);
```

其中参数 `work` 指向被初始化的工作，参数 `func` 是工作函数指针。另外一个接口函数可以单独修改工作函数，其他成员不变，其原型如下：

```
void PREPARE_WORK(struct work_struct *work, work_func_t func);
```

其中参数 `work` 指向被修改的工作，参数 `func` 是新的工作函数指针。

工作也可以直接定义并初始化，用下面这个宏：

```
DECLARE_WORK(name, func);
```

其中参数 `name` 是所定义的工作的变量名，参数 `func` 是工作函数指针。

14.3.3.2 工作队列的创建与销毁

工作队列的创建包含两个内容：一是创建用于容纳工作的数据结构，二是创建专门用于调度各个工作的内核线程。创建工作队列的接口函数原型如下：

```
struct workqueue_struct *create_workqueue(const char *name);
```

其参数及返回值的含义解释如下。

- ◆ `name`：字符串，新创建的工作队列的名称。
- ◆ 返回值：指向新创建的工作队列，如果创建失败，则返回 `NULL`。

内核将为新的工作队列创建一系列新的内核线程，称为工作线程。如果工作队列的名称是 `wq`，则对应的工作线程的名称就是 `wq/%`，其中 `%` 是一个数字，代表 CPU 的编号。每个工作线程都绑定在一个 CPU 上，专门负责执行在这个 CPU 上入队的工作，因此同一个工作队列中的工作有可能同时在多个 CPU 上运行。如果不需要这个特征，则可使用以下函数：

```
struct workqueue_struct *create_singlethread_workqueue(const char *name);
```

这个函数的用法与 `create_workqueue` 相同，功能也类似，但是它只为新的工作队列创建一个工作线程，线程的名称与工作队列的名称相同。这个工作线程与 CPU 没有绑定关系，它在哪个 CPU 上得到执行机会，就会执行工作队列里那些在这个 CPU 上入队的工作。由于只有一个工作线程，因此同一个工作队列内的工作不可能同时在多个 CPU 上执行。

每创建一个工作队列都会消耗系统资源，不用的时候应该将其销毁以释放资源。销毁工作队列的接口函数原型如下：

```
void destroy_workqueue(struct workqueue_struct *wq);
```

其中参数 **wq** 指向要被销毁的工作队列。工作队列被销毁时，其数据结构所占内存被释放，对应的内核线程终止，但在这之前会等待已在队列里的工作完成。

14.3.3.3 工作的入队与撤销

有了工作队列之后，所定义的工作就可以放到队列里等候执行了。将工作放到工作队列里的接口函数原型如下：

```
int queue_work(struct workqueue_struct *wq, struct work_struct *work);
```

其各个参数及返回值的含义解释如下。

- ◆ **wq**：指向工作要放入的工作队列。
- ◆ **work**：指向要入队的工作。
- ◆ 返回值：非 0 表示入队成功，0 表示入队失败，这个工作本来就已经在某个工作队列内。

入队的工作将与使它入队的代码在同一 CPU 上执行，执行时处在相应的内核工作线程的上下文内，因此工作中可以有进程调度操作。但由于是在内核线程中，所以访问用户态内存的操作仍然不可行。

工作执行完成后就会自动离开工作队列，所以入队的工作只执行一次，除非在它的工作函数内又将自身重新入队。在工作执行之前，还可以用下面的函数将工作撤销：

```
int cancel_work_sync(struct work_struct *work);
```

其参数及返回值的含义解释如下。

- ◆ **work**：指向被撤销的工作。
- ◆ 返回值：非 0 表示撤销成功，0 表示撤销失败，工作本来就不在队列内。

如果工作已经在执行，则这个函数会等待它执行完成。

14.3.3.4 工作队列的冲尽

内核提供了一些接口函数，可以等待工作队列内的工作执行完成，到函数返回时，工作队列内的指定工作全部执行完毕离开队列，本书将类似这样的操作称为冲尽。

下面这个函数可以冲尽工作队列内现有的全部工作：

```
void flush_workqueue(struct workqueue_struct *wq);
```

但如果在等待期间又有新的工作入队，则新的工作不予考虑。其中参数 **wq** 指向要冲尽的工作队列。下面这个函数可以冲尽到一个指定的工作：

```
int flush_work(struct work_struct *work);
```

其参数及返回值的含义解释如下。

- ◆ **work**：指向要冲尽的工作。

◆ 返回值：非 0 表示成功，0 表示失败，工作不在队列中。

由于入队的工作是按顺序执行的，所以要冲尽到指定的工作，排在它前面的工作首先要执行完毕。这里要注意的是，如果有多个 CPU，则在各个 CPU 上入队的工作是分别排队的。

14.3.3.5 判断状态

通过工作的状态位可以直接判断工作是否处在队列中待执行，对此，内核提供了一个接口函数，原型如下：

```
int work_pending(struct work_struct *work);
```

其参数及返回值的含义解释如下。

- ◆ work：指向被判断的工作。
- ◆ 返回值：非 0 表示在队列内，0 表示不在队列内。

14.3.3.6 使用内核工作队列

多数情况下，驱动不需要自己创建工作队列，而是将工作入队到内核自己创建的工作队列上，使用的函数接口原型如下：

```
int schedule_work(struct work_struct *work);
```

其参数及返回值的含义解释如下。

- ◆ work：指向要入队的工作。
- ◆ 返回值：非 0 表示入队成功，0 表示入队失败，这个工作本来就已经在某个工作队列内。

还有一个函数可以冲尽内核的工作队列，原型如下：

```
void flush_scheduled_work(void);
```

内核工作队列的名称为 **event**。使用内核工作队列的时候要注意，由于它是所有驱动共享的，一个执行时间过长的工作会影响其他驱动中后续入队工作的执行。

14.3.3.7 延时工作

如果需要一个工作过一段时间后再开始执行，则可以将它与一个定时器结合。内核已经包装了相应的接口供编程使用，称为延时工作。延时工作的数据类型定义如下：

```
struct delayed_work {
    struct work_struct work; /* 工作 */
    struct timer_list timer; /* 定时器 */
};
```

相应地也有一套用于操作延时工作的接口函数。

初始化延时工作的接口如下：

```
void INIT_DELAYED_WORK(struct delayed_work *dwork, work_func_t func);
```

这个函数的功能是初始化参数 `dwork` 指向的延时工作，参数 `func` 是工作函数指针。
修改延时工作函数的接口如下：

```
void PREPARE_DELAYED_WORK(struct delayed_work *dwork, work_func_t func);
```

这个函数的功能是修改参数 `dwork` 指向的延时工作的工作函数指针为 `func`。
直接定义并初始化延时工作的接口如下：

```
DECLARE_DELAYED_WORK(name, func);
```

其中参数 `name` 是所定义的延时工作的变量名，参数 `func` 是工作函数指针。

需要注意的是，延时工作的工作函数类型也是 `work_func_t` 型，这个函数的参数是 `(struct work_struct *)` 型，显然它指向延时工作的 `work` 成员，可以用内核提供的如下函数转换成指向延时工作自己的指针：

```
static inline struct delayed_work *to_delayed_work(struct work_struct *work)
{
    return container_of(work, struct delayed_work, work);
}
```

这里又用到了 `container_of` 宏。

延时工作入队的接口如下：

```
int queue_delayed_work(struct workqueue_struct *wq,
    struct delayed_work *dwork, unsigned long delay);
```

其各个参数及返回值的含义解释如下。

- ◆ `wq`：指向延时工作要放入的工作队列。
- ◆ `dwork`：指向要入队的延时工作。
- ◆ `delay`：延时的时间，单位是 `jiffies`。注意这里的 `delay` 是延时的时间，定时器的过期时间将被设为当前时间加上 `delay`。
- ◆ 返回值：非 0 表示入队成功，0 表示入队失败。

使延时工作进入内核工作队列的接口如下：

```
int schedule_delayed_work(struct delayed_work *dwork, unsigned long delay);
```

这个函数的功能类似于 `queue_delayed_work`，只是将延时工作入队到内核的工作队列上。
撤销延时工作的接口如下：

```
int cancel_delayed_work_sync(struct delayed_work *dwork);
```

这个函数的功能是撤销参数 `dwork` 指向的延时工作。返回非 0 表示撤销成功，0 表示撤销失败。

判断延时工作是否在队列中：

```
int delayed_work_pending(struct delayed_work *dwork);
```

这个函数的功能是判断参数 `dwork` 指向的延时工作是否在队列中。返回非 0 表示在队列中，0 表示不在队列中。

14.3.4 定时器例程

在这里我们将编写一个使用定时器和等待队列的例程。这个例程实现了一个字符设备，它的功能是：应用程序读设备文件时将得到一个字符串，这个字符串由一个定时器控制，每隔一秒重新生成一次，其内容包含了定时器的过期时间和当前时间等。

14.3.4.1 源码

例程源代码如下：

```
/* 文件名: kgrunt.c */
/* 说明: 定时器例程 */

#include <linux/module.h> /* 模块编程 */
#include <linux/fs.h> /* 设备号、文件操作接口 */
#include <linux/cdev.h> /* 字符设备编程接口 */
#include <linux/uaccess.h> /* 访问用户态内存接口 */
#include <linux/timer.h> /* 定时器接口 */
#include <linux/sched.h> /* 进程调度相关 */
#include <linux/wait.h> /* 等待队列接口 */

MODULE_LICENSE("GPL"); /* 版权声明 */

#define BUF_SIZE 256 /* 内核缓冲区的大小 */

/* 以下类型表示 kgrunt 设备 */
struct kgrunt_dev {
    struct cdev cdev; /* 内嵌一个字符设备作为代表向内核注册 */
    char *buf; /* 保存内核缓冲区的首地址 */
    int len; /* 保存内核缓冲区的长度 */
    int bytes; /* 保存内核缓冲区中字符的个数 */
    struct timer_list timer; /* 定时器 */
    int timer_flag; /* 控制是否继续注册定时器 */
    wait_queue_head_t queue; /* 读操作等待队列 */
    dev_t dev; /* 保存对应的设备号 */
};

static void kgrunt_timer_fn(unsigned long d)
{
    /* 从参数得到 kgrunt 设备的地址 */
    struct kgrunt_dev *dev = (struct kgrunt_dev *)d;
    /* 得到定时器变量的地址 */
    struct timer_list *timer = &dev->timer;
    /* 输出字符串到内核缓冲区 */
    dev->bytes = scnprintf(dev->buf, dev->len,
        "timer expires: %lu, jiffies: %lu, current pid: %d, comm: %s.\n",
        timer->expires, jiffies, current->pid, current->comm);
}
```

```

/* 唤醒等待读的进程 */
wake_up_interruptible(&dev->queue);
/* 如果还需要继续注册定时器，则继续注册，过期时间推后一秒 */
if (dev->timer_flag) mod_timer(timer, timer->expires+HZ);
}

/* 打开操作 */
static int kgrunt_open(struct inode *inode, struct file *file)
{
    /* 由 inode 对应的字符设备指针得到 kgrunt 设备的地址 */
    struct kgrunt_dev *dev =
        container_of(inode->i_cdev, struct kgrunt_dev, cdev);
    pr_debug("kgrunt_open: be called!\n");
    /* 将 kgrunt 设备的地址保存在打开的文件中 */
    file->private_data = dev;
    /* 注册定时器，如是已经（被其他进程）注册，则修改它 */
    mod_timer(&dev->timer, jiffies+HZ); /* 定时器将于一秒后触发 */
    dev->timer_flag = 1; /* 设置标志，让定时器持续注册自身 */
    /* 返回 0 表示打开成功 */
    return 0;
}

/* 关闭操作 */
static int kgrunt_release(struct inode *inode, struct file *file)
{
    /* 由 inode 对应的字符设备指针得到 kgrunt 设备的地址 */
    struct kgrunt_dev *dev =
        container_of(inode->i_cdev, struct kgrunt_dev, cdev);
    pr_debug("kgrunt_release: be called!\n");
    /* 先将继续注册定时器的标志清零，再注销定时器，否则在对称多处理器平台上，
     * 当 del_timer_sync 返回时可能定时器已经再次注册上 */
    dev->timer_flag = 0;
    /* 注销定时器 */
    del_timer_sync(&dev->timer);
    return 0;
}

/* 读操作 */
static ssize_t kgrunt_read(struct file *file, char __user *buf,
    size_t count, loff_t *pos)
{
    int err; /* 用于保存错误码 */
    /* 从 file 中得到对应的 kgrunt 设备 */
    struct kgrunt_dev *dev = (struct kgrunt_dev *)file->private_data;
    pr_debug("kgrunt_read: pos = %d, count = %d.\n", (int)*pos, count);
    /* 参数合法性检查 */
    if (count < 0) return -EINVAL; /* 非法参数 */
    if (count == 0) return 0; /* 直接返回 0 */
    if (dev->bytes == 0) { /* 无数据可读 */
        /* 非阻塞方式下返回 -EAGAIN */
        if (file->f_flags & O_NONBLOCK) return -EAGAIN;
    }
}

```

```

    /* 等待直到缓冲区不空 */
    err = wait_event_interruptible(dev->queue, dev->bytes > 0);
    /* 发生信号则返回 -EINTR */
    if (err == -ERESTARTSYS) return -EINTR;
}
/* 控制读取的长度, 不能超过当前有效数据的个数 */
if (count > dev->bytes) count = dev->bytes;
/* 从内核缓冲区复制数据到 buf 指向的用户态内存。
 * 让每次读都从缓冲区开始复制数据, 因此读写位置无意义 */
if (copy_to_user(buf, dev->buf, count) > 0) {
    pr_debug("kgrunt_read: copy_to_user() FAILED!\n");
    return -EFAULT;
}
/* 读取之后将缓冲区中的数据清空 */
dev->bytes = 0;
/* 返回读到的字节数 */
return count;
}

/* 写操作 */
static ssize_t kgrunt_write(struct file *file, const char __user *buf,
                           size_t count, loff_t *pos)
{
    /* 不支持写操作, 直接返回错误码 */
    return -EPERM;
}

/* 定位操作 */
static loff_t kgrunt_llseek(struct file *file, loff_t offset, int whence)
{
    /* 不支持定位操作, 直接返回错误码 */
    return -EPERM;
}

/* 设备的文件操作, 全局变量 */
static struct file_operations kgrunt_fops = {
    .owner = THIS_MODULE, /* 所属模块 */
    .open = kgrunt_open, /* 打开操作 */
    .release = kgrunt_release, /* 关闭操作 */
    .read = kgrunt_read, /* 读操作 */
    .write = kgrunt_write, /* 写操作 */
    .llseek = kgrunt_llseek, /* 定位操作 */
};

/* 全局变量, 表示一个 kgrunt 设备 */
static struct kgrunt_dev kgrunt;

/* 模块的初始化函数 */
static __init int kgrunt_init(void)
{
    int err; /* 用于保存错误码 */

```

```

pr_debug("kgrunt_init: be called!\n");
/* 分配设备号 */
if ((err = alloc_chrdev_region(&kgrunt.dev, 0, 1, "kgrunt")) < 0) {
    pr_debug("kgrunt_init: alloc_chrdev_region() ERR = %d!\n", -err);
    goto alloc_chrdev_region_fail;
}
pr_debug("kgrunt_init: dev = (%d, %d).\n",
        MAJOR(kgrunt.dev), MINOR(kgrunt.dev));
/* 分配内核缓冲区 */
if ((kgrunt.buf = kmalloc(BUF_SIZE, GFP_KERNEL)) == NULL) {
    pr_debug("kgrunt_init: kmalloc() ERR!\n");
    err = -ENOMEM; /* 此错误号表示内存不足 */
    goto kmalloc_kgrunt_buf_fail;
}
/* 保存缓冲区的长度 */
kgrunt.len = BUF_SIZE;
kgrunt.bytes = 0;
/* 初始化字符设备 */
cdev_init(&kgrunt.cdev, &kgrunt_fops);
kgrunt.cdev.owner = kgrunt_fops.owner;
/* 初始化定时器 */
setup_timer(&kgrunt.timer, kgrunt_timer_fn, (unsigned long)&kgrunt);
kgrunt.timer_flag = 0;
/* 初始化等待队列 */
init_waitqueue_head(&kgrunt.queue);
/* 注册字符设备 */
if ((err = cdev_add(&kgrunt.cdev, kgrunt.dev, 1)) < 0) {
    pr_debug("kgrunt_init: cdev_add() ERR = %d!\n", -err);
    goto cdev_add_fail;
}
return 0; /* 返回 0 表示初始化成功 */
/* 以下为出错处理，各个清理操作按初始化时的倒序排列，出错时跳转到相应的位置，
 * 这是驱动编程中常用的做法 */
cdev_add_fail:
    kfree(kgrunt.buf); /* 释放所分配的内存 */
kmalloc_kgrunt_buf_fail:
    unregister_chrdev_region(kgrunt.dev, 1); /* 注销所注册的设备号 */
alloc_chrdev_region_fail:
    return err;
}
module_init(kgrunt_init); /* 指定 kgrunt_init 为模块的初始化函数 */

/* 模块的退出函数 */
static __exit void kgrunt_exit(void)
{
    pr_debug("kgrunt_exit: be called!\n");
    /* 模块退出时进行清理，按初始化时的倒序操作 */
    cdev_del(&kgrunt.cdev); /* 注销字符设备 */
    kfree(kgrunt.buf); /* 释放所分配的内存 */
    unregister_chrdev_region(kgrunt.dev, 1); /* 注销所注册的设备号 */
}

```



```
module_exit(kgrunt_exit); /* 指定 kgrunt_exit 为模块的退出函数 */
```

14.3.4.2 说明

定时器的注册操作放在了设备的打开操作中，相应的注销操作放在了关闭操作中。当应用程序打开设备文件时，定时器就被注册，并将于一秒后触发。

当应用程序进行读操作时，如果是阻塞方式且内核缓冲区没有数据，它将进入设备的等待队列。在定时器的工作函数中，数据被写入内核缓冲区并且将等待的进程唤醒。

为了简单，每次进行读操作时，数据总是从缓冲区的开始处读取，因此对应用程序来说没有读写位置的概念。

本例程在 Linux 2.6.26 和 Linux 2.6.30 内核上编译并测试通过。

14.4 自旋锁与同步

自旋锁是内核特有的一种同步手段，它的实现必须依赖于 CPU 本身的支持。自旋锁主要用于 SMP 平台上多个 CPU 之间的同步。可以说，如果没有自旋锁，Linux 不可能支持 SMP 平台。

内核中也常用一些与应用编程中类似的同步手段，如互斥体和信号灯等，这些编程接口也将在本节中介绍。

14.4.1 并发与竞态

在 Linux 操作系统上，多个进程可以同时运行，此外各种不断发生的中断也在同时得到处理，这种多个上下文宏观上同时运行的情况称为并发。如果仔细考虑，并发有以下几种可能性。

- ◆ UP 平台上，一个进程正在执行时被另一个进程抢占。
- ◆ UP 平台上，一个进程正在执行时发生了中断，内核转而执行中断处理程序。
- ◆ SMP 平台上，每个处理器上都会发生 UP 平台上的情况。
- ◆ SMP 平台上，多个进程或中断同时在多个 CPU 上执行。

并发本身并不造成问题。但如果考虑一种极端情况，两个进程同时访问同一块内存区域时，有可能造成操作的不连续，最后导致错误的结果。

举个例子，分析 14.2.7 节中 kpipe 例程的读写操作的代码。如果没有同步代码，由于并发的存在，当进程甲开始做读写操作并且已经计算好了要读写的字节数时，进程乙可能也在做读写操作并且修改了缓冲区头尾指针的值，这样，当进程甲开始读写缓冲区时，算好的字节数已不符合缓冲区的实际情况，可能造成内存访问越界。

这种多个并发的上下文同时使用同一个资源的情况称为竞态，而可能发生问题的这一段代码称为临界区。内核编程时的临界区属于以下两种情况的较多。

- ◆ 代码中访问了全局变量，并且这段代码可被多个进程调用。
- ◆ 两段代码中访问了相同的全局变量，并且一段代码被进程调用，另一段属于中断处理程序。



局部变量位于各个进程的栈上，不同的上下文访问的是不同的内存单元，不会造成竞态。

竞态可以通过以下手段加以解决。

一、同步

同步的本质是让一个执行流程停滞在某处，以等待另外一个执行流程到达指定的状态。如果在进入和离开临界区时加上同步操作，使试图进入临界区的执行流程一直等待，直到先进入临界区的执行流程离开临界区后再继续，就可以避免多个执行流程同时执行临界区代码的情况。

二、原子操作

如果一个操作本身在执行上就是不可分割的，则不会有竞态问题，这种操作称为原子操作。

14.4.2 自旋锁

自旋锁是内核最根本的同步手段。对于自旋锁有两个基本操作：获取和释放。获取操作可由图 14.4 解释。自旋锁本身用一个变量表示，变量至少有两个取值，表示锁的两个状态：未锁和已锁。获取自旋锁时，首先读取锁的状态，如果是未锁状态，则马上将其改为已锁状态，完成获取操作；如果是已锁状态，则再次读取锁的状态进行判断。当锁一直在已锁状态时，CPU 实际上在执行一个循环。注意这里的循环是一个“忙等待”的过程，中间没有任何进程调度操作。

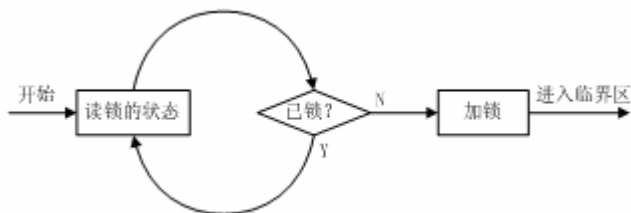


图 14.4 获取自旋锁操作示意

释放锁则是一个相反的操作，将自旋锁的状态改为未锁。释放后，另外一个试图获取同一把锁并陷入循环的执行流程就可以继续执行并获得锁。显然，试图获取锁并陷入循环的执行流程不可能自己结束循环，只能等待其他执行流程释放这把锁。

获取自旋锁时，当判断锁的状态为未锁但还没有加锁时，如果有其他执行流程也在这时判断锁的状态是未锁，结果两个执行流程将一起进入临界区。因此，从判断锁的状态为未锁到加锁的过程必须是原子操作，不允许任何其他执行流程打破操作的连续性。在 SMP 平台上，CPU 必须有相应的指令来支持基本操作的原子性。

14.4.2.1 抢占的禁用与使能

禁用抢占获取自旋锁的先决条件，否则当一个进程获取锁并进入临界区后，如果被另外一个进程抢占并试图获取同一把锁，CPU 将陷入死锁状态。内核提供了禁用和使能抢占的接口，定义在以下头文件中：

```
#include <linux/preempt.h>
```

禁用抢占的接口函数如下：

```
void preempt_disable(void);
```

获取自旋锁时禁用抢占，对应地，释放自旋锁时要使能抢占，接口函数如下：

```
void preempt_enable(void);
```

这些接口实际上都是宏，在非抢占式内核上，它们都被定义为无操作。



抢占的禁用和使能都是有计数的，如果被禁用两次，则必须使能两次才能回到原来的状态。

14.4.2.2 中断的禁用与使能

如果要解决进程和中断的竞态问题，还需要禁用中断，否则当一个进程获取锁并进入临界区时，如果发生了中断且中断中又在获取同一把锁，CPU 将陷入死锁状态。内核提供了禁用和使能中断的接口，接口的实现是平台相关的，使用时需要包含以下头文件：

```
#include <linux/irqflags.h>
```

在当前 CPU 上禁用中断的接口函数如下：

```
void local_irq_disable(void);
```

注意它只是让当前 CPU 不处理任何中断，其他 CPU 仍然保持原来的中断状态。

在当前 CPU 上使能中断的接口函数如下：

```
void local_irq_enable(void);
```

只使用这两个接口函数还不能完全解决问题。考虑这样的情况：在进入临界区之前要做中断禁用操作，如果在禁用之前中断已经处于禁用状态，那么离开临界区时使能中断的操作就是不合理的。解决这个问题可以用另外两个接口，其一是保存当前中断状态并禁用中断：

```
void local_irq_save(unsigned long flags);
```

其中参数 flags 用来保存当前中断状态。注意这个接口实际上是一个宏，因此参数 flags 不需要传递指针也能被修改。对应的操作是恢复中断状态到已保存的值：

```
void local_irq_restore(unsigned long flags);
```

这样，不管原来的中断状态是什么，都能在禁用后正确地将其设置回原来的值。

以上接口操作的都是 CPU 本身的中断标志位，禁用后所有中断都不能发生。一般来说，对应于每个中断号还有独立的屏蔽位设置，因此用以上接口使能中断后，并不意味着所有中断都可以发生。

14.4.2.3 自旋锁的定义与初始化

自旋锁的实现也是平台相关的。但在使用时，只需统一包含以下头文件：

```
#include <linux/spinlock.h>
```

自旋锁变量的类型是 `spinlock_t`，定义如下：

```
typedef struct {  
    volatile unsigned int lock;  
} raw_spinlock_t;  
  
typedef struct {  
    raw_spinlock_t raw_lock;  
} spinlock_t;
```

可见自旋锁的状态是用一个无符号整数表示的。这是在 **SMP** 平台上的定义，如果内核编译成 **UP** 平台版本，则 `raw_spinlock_t` 类型的定义退化为：

```
typedef struct { } raw_spinlock_t;
```

可见在内核的 **UP** 版本中，`spinlock_t` 实际上是一个空的数据类型，当然也不可能保存锁的状态。这是因为在 **UP** 平台上，获取锁的循环操作是无意义的，一旦执行流程陷入循环，它就会永远循环下去，所以在实现上进行了简化，不判断锁的状态直接进入临界区，锁的状态也就失去了意义。



大多数 Linux 发行版不管是装在 **SMP** 平台上，还是装在 **UP** 平台上，用的都是 **SMP** 版本的内核。

定义的自旋锁变量必须先初始化才能使用，初始化的接口函数如下：

```
void spin_lock_init(spinlock_t *lock);
```

其中参数 `lock` 指向要初始化的自旋锁变量。

也可以直接定义并初始化一个自旋锁：

```
DEFINE_SPINLOCK(name); /* 定义自旋锁 name 并将其初始化 */
```

14.4.2.4 自旋锁的获取与释放

获取自旋锁的接口函数原型如下：

```
void spin_lock(spinlock_t *lock);
```

其中参数 `lock` 指向要获取的自旋锁。

获取自旋锁时首先要禁止抢占，然后开始循环判断锁的状态。在内核的 **UP** 版本中，它唯一的操作就是禁止抢占，如果是 **UP** 版本且非抢占式内核，则进一步退化为无操作。

释放自旋锁的接口函数如下：

```
void spin_unlock(spinlock_t *lock);
```

其中参数 `lock` 指向要释放的自旋锁。

释放自旋锁时首先要使能抢占，然后将锁的状态置为未锁。在内核的 **UP** 版本中，它唯一的

操作就是使能抢占，如果是 UP 版本且非抢占式内核，则进一步退化为无操作。

以上为自旋锁的基本操作。内核还提供了一个接口函数，可以尝试获取自旋锁，如果锁的状态是已锁也不会陷入循环，它的原型如下：

```
int spin_trylock(spinlock_t *lock);
```

其中参数 **lock** 指向尝试获取的自旋锁，当返回值非 0 时表示获取成功，返回值是 0 时表示获取失败，自旋锁原来就在已锁状态。对于 UP 版本的内核，由于没有真正的获取自旋锁操作，它的作用就是直接禁用抢占并返回 1。



如果使用 **spin_trylock** 接口获取自旋锁成功，则最终也要用 **spin_unlock** 接口释放；如果没有获取成功，则不必释放。

还有一个接口函数可以等待自旋锁的解锁，但并不获取它，其原型如下：

```
void spin_unlock_wait(spinlock_t *lock);
```

其中参数 **lock** 指向要等待的自旋锁。注意等待过程中没有进程调度，是一个“忙等待”。

14.4.2.5 与中断互斥

以上接口并没有考虑在获取锁以后又发生中断的问题，如果要解决与中断的互斥问题，则需要用下面介绍的接口。

禁止中断并获取自旋锁的函数如下：

```
void spin_lock_irq(spinlock_t *lock);
```

其中参数 **lock** 指向要获取的自旋锁。

释放自旋锁并使能中断的函数如下：

```
void spin_unlock_irq(spinlock_t *lock);
```

其中参数 **lock** 指向要释放的自旋锁。

禁止中断并保存中断状态，然后获取自旋锁的函数如下：

```
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
```

其中参数 **lock** 指向要获取的自旋锁，参数 **flags** 用于保存中断状态。

释放自旋锁并将中断状态恢复为已保存的值的函数如下：

```
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
```

其中参数 **lock** 指向要释放的自旋锁，参数 **flags** 是已保存的中断状态。

禁止中断并尝试获取自旋锁的函数如下：

```
int spin_trylock_irq(spinlock_t *lock);
```

其中参数 **lock** 指向尝试获取的自旋锁，返回值非 0 表示获取成功，返回值为 0 表示获取失败。如果获取失败，则中断状态恢复为使能状态。

禁止中断并保存中断状态，然后尝试获取自旋锁的函数如下：

```
int spin_trylock_irqsave(spinlock_t *lock, unsigned long flags);
```

其中参数 `lock` 指向尝试获取的自旋锁，参数 `flags` 用于保存中断状态，返回值非 0 表示获取成功，返回值为 0 表示获取失败。如果获取失败，则中断状态恢复为原来的状态。

由于抢占的实现依赖于中断，如果中断被禁用，则抢占自然也不能发生，因此以上操作中不需要再禁用抢占。

关于自旋锁的各种接口可总结如表 14.1 所示。

表 14.1 自旋锁操作接口

操作	进程间互斥	与中断互斥	与中断互斥并保存中断状态
获取	<code>spin_lock</code>	<code>spin_lock_irq</code>	<code>spin_lock_irqsave</code>
尝试获取	<code>spin_trylock</code>	<code>spin_trylock_irq</code>	<code>spin_trylock_irqsave</code>
释放	<code>spin_unlock</code>	<code>spin_unlock_irq</code>	<code>spin_unlock_irqrestore</code>

14.4.2.6 状态判断

内核还提供了一个用于判断自旋锁状态的函数：

```
int spin_can_lock(spinlock_t *lock);
```

其参数及返回值的含义解释如下。

- ◆ `lock`：指向要判断的自旋锁。
- ◆ 返回值：0 表示自旋锁为已锁状态，可被获取；非 0 表示未锁状态，不可被获取。

这个函数的使用价值并不大，因为即使判断一个锁的状态为未锁，在进一步操作之前，它也有可能已被其他执行流程获取，成为已锁状态。

14.4.2.7 自旋锁的使用原则

自旋锁是内核的一种特殊的同步方式，是实现其他所有同步接口的基础。它在使用上一般要遵循以下原则。

- ◆ 能不用尽量不用，持有锁的时间应尽可能短。这是因为持有锁以后将使其他 CPU 上要获取同一把锁的执行流程陷入空循环，消耗 CPU 资源。
- ◆ 持有锁以后尽量不要再去获取另一把锁，如果需要，则代码各处获取锁的顺序应一致，否则容易引起死锁。
- ◆ 从性能优先的角度考虑，如果不需要与中断互斥，则不要使用禁止中断的接口。

14.4.3 原子上下文

当一个进程持有自旋锁以后，内核认为它进入了一种特殊状态，这时，任何形式的进程调度都是不允许的，这种上下文状态称为原子上下文。中断上下文中本身就不允许进程调度的发生，因此

也属于原子上下文。

在原子上下文中，任何可能直接或间接引起进程调度的操作都是不允许的。这些操作大体上有以下这些。

- ◆ 调度：schedule。
- ◆ 分配内存：kmalloc, kcalloc, 当标志不是 GFP_ATOMIC 时。
- ◆ 等待队列的等待操作：wait_event 等。
- ◆ 访问用户态内存：copy_from_user, copy_to_user。
- ◆ tasklet 操作：tasklet_kill。
- ◆ 工作队列的创建和冲尽：create_workqueue, flush_workqueue, flush_work 等。



由于内核的编程接口繁多，且没有统一的文档描述，所以在原子上下文中误用引起进程调度的接口是一种很容易犯的错误。

当一段临界区代码用自旋锁保护起来以后，可以称这段代码所做的操作是原子的，也就是说，操作有了不可分割性，不可能中途被其他执行流程打断。这里需要注意，是否是原子操作也有相对性：如果只是进程间有竞态，那么用 spin_lock 保护起来的代码就可以认为是原子的，尽管在操作过程中还有可能发生中断；如果是进程与中断间有竞态，那就必须用 spin_lock_irq 或 spin_lock_irqsave 保护起来才能成为原子操作。

代码的执行流程处于何种上下文并非只是一些概念，实际上内核中有相应的标志来表示各种上下文状态，这些状态可以通过一些接口获得。

下面的函数可以判断当前上下文是否在硬中断内：

```
int in_irq(void);
```

如果当前上下文在硬中断内，这个函数返回非 0，否则返回 0。

下面的函数可以判断当前上下文是否在软中断内：

```
int in_softirq(void);
```

如果当前上下文在软中断内，这个函数返回非 0，否则返回 0。

下面的函数可以判断当前上下文是否在中断内：

```
int in_interrupt(void);
```

如果当前上下文在中断内，这个函数返回非 0，否则返回 0。可以认为这个函数等价于 in_irq() || in_softirq()。

下面的函数可以判断当前上下文是否在原子上下文内：

```
int in_atomic(void);
```

如果当前上下文在原子上下文内，这个函数返回非 0，否则返回 0。

这些函数定义在头文件 <linux/hardirq.h> 中，进行驱动编程时一般不会用到。

14.4.4 读写锁

多个执行流程同时对同一块内存进行只读的访问不会造成问题。考虑这样一种情况：并发的多个执行流程进入临界区后主要进行读操作，只有很小的概率会进行写操作，这时如果使用自旋锁进行互斥，则在一个执行流程进入临界区后，其他本来可以安全地进行读操作的执行流程都会陷入循环，这是 CPU 资源的巨大浪费。为了解决这种问题，内核提供了读写锁的接口。读写锁的实现原理与自旋锁相似，它的相关定义与声明也放在以下头文件中：

```
#include <linux/spinlock.h>
```

对读写锁的获取和释放操作都被分为两种：读锁获取和写锁获取。它们的特点如下。

- ◆ 当写锁被获取后，任何获取锁的操作都不能成功。
- ◆ 当读锁被获取后，获取写锁的操作将失败，但获取读锁的操作仍能成功。

也就是说，使用读写锁时，允许多个执行流程同时获取读锁，但只能有一个执行流程获取写锁，且写锁与读锁也是互相排斥的。

14.4.4.1 读写锁的定义与初始化

在 SMP 版本的内核上，读写锁的定义如下：

```
typedef struct {  
    volatile unsigned int lock;  
} raw_rwlock_t;  
  
typedef struct {  
    raw_rwlock_t raw_lock;  
} rwlock_t;
```

可见它的定义与自旋锁的定义完全相同。实际上，内核是这样来实现读写锁的。

- ◆ 初始化时，将读写锁中的整型变量赋一个足够大的值。
- ◆ 获取读锁时，如果这个变量的值不为 0，说明可以获取，获取后它的值会减 1。
- ◆ 获取写锁时，如果这个变量的值与初始值一样大，说明可以获取，获取后它的值减到 0。
- ◆ 释放读锁时，这个变量的值加 1。
- ◆ 释放写锁时，这个变量的值加回初始值。

而在 UP 版本的内核上，raw_rwlock_t 类型的定义同样为空：

```
typedef struct { } raw_rwlock_t;
```

定义的读写锁在使用前必须初始化，可用以下函数：

```
void rwlock_init(rwlock_t *lock);
```

其中参数 lock 指向被初始化的读写锁。

也有一个宏可以直接定义读写锁并初始化：


```
DEFINE_RWLOCK(name); /* 定义读写锁，变量名为 name，并将其初始化 */
```

14.4.4.2 读写锁的获取与释放

与自旋锁相比较，读写锁对应的接口函数的用法和功能都类似，但几乎全部分裂为“读”和“写”两个版本。

获取读锁：

```
void read_lock(rwlock_t *lock);
```

获取写锁：

```
void write_lock(rwlock_t *lock);
```

释放读锁：

```
void read_unlock(rwlock_t *lock);
```

释放写锁：

```
void write_unlock(rwlock_t *lock);
```

尝试获取读锁：

```
int read_trylock(rwlock_t *lock);
```

尝试获取写锁：

```
int write_trylock(rwlock_t *lock);
```

14.4.4.3 与中断互斥

禁用中断并获取读锁：

```
void read_lock_irq(rwlock_t *lock);
```

禁用中断并获取写锁：

```
void write_lock_irq(rwlock_t *lock);
```

释放读锁并使能中断：

```
void read_unlock_irq(rwlock_t *lock);
```

释放写锁并使能中断：

```
void write_unlock_irq(rwlock_t *lock);
```

禁用中断且保存中断状态，然后获取读锁：

```
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
```

禁用中断且保存中断状态，然后获取写锁：

```
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
```



释放读锁，然后将中断状态恢复为已保存状态：

```
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
```

释放写锁，然后将中断状态恢复为已保存状态：

```
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
```

禁用中断且保存中断状态，然后尝试获取写锁：

```
int write_trylock_irqsave(rwlock_t *lock, unsigned long flags);
```

14.4.4.4 状态判断

判断读锁是否可以被获取：

```
int read_can_lock(rwlock_t *rwlock);
```

这个函数返回 0 表示读锁不能被获取，说明写锁已被获取，返回非 0 表示读锁可以被获取。

判断写锁是否可以被获取：

```
int write_can_lock(rwlock_t *rwlock);
```

这个函数返回 0 表示写锁不能被获取，说明写锁或读锁已被获取，返回非 0 表示写锁可以被获取。

14.4.5 原子类型

在内核中大量地用到类似计数器的变量，这些变量只做简单的加减法和判断操作。通常，这些简单的操作都可以用一条 CPU 指令实现。如果考虑 SMP 平台且 CPU 内部有缓存的情况，则即使是一条 CPU 指令所做的操作也不能认为是原子的。因为缓存与内存之间的数据同步并不是在每次执行读写内存指令时都会发生。设想这样的情况：当被一个 CPU 修改的变量尚未保存回内存时，另一个 CPU 就使用了这个变量的值，则会造成数据不一致的情况。另外，有些 CPU 没有直接对内存单元进行加减操作的指令，需要先将数据读到寄存器，修改之后再保存回内存，实现上就是多条指令，当然更不是原子操作。

自旋锁的实现上已经考虑了这些问题，可以有效地将这些操作变为原子操作，如：

```
DEFINE_SPINLOCK(my_lock); /* 定义自旋锁 my_lock */
int count = 0; /* 计数器变量 */
void inc_count(void) /* 假定这个函数会被多个进程同时调用 */
{
    spin_lock(&my_lock); /* 获取自旋锁 */
    count++; /* 使计数器加 1 */
    spin_unlock(&my_lock); /* 释放自旋锁 */
}
```

但是对于一个如此简单的操作，使用自旋锁的开销显得太大。实际上，CPU 本身就有特殊的指令可以保证简单操作的原子性，内核把这些指令都封装成了接口函数，使用时需要包含以下头文件：

```
#include <asm/atomic.h>
```



14.4.5.1 定义与初始化

这些操作都作用于一种特定的数据类型，称为原子类型，定义如下：

```
typedef struct {  
    volatile int counter;  
} atomic_t;
```

可见原子类型的变量实质上就等价于整型数。

原子型变量可以用以下方式初始化：

```
atomic_t count = ATOMIC_INIT(0); /* 将原子变量 count 的初值设为 0 */
```

用以下函数可以给原子变量赋值：

```
void atomic_set(atomic_t *v, int i);
```

这个函数的功能是将 `v` 指向的原子变量赋值为 `i`。

用下面的函数可以读取原子变量的值：

```
int atomic_read(atomic_t *v);
```

这个函数的功能是返回 `v` 指向的原子变量的值。

必须注意，为了保持操作的原子性，即使是赋值和取值操作，也应该使用这些函数，而不是直接使用赋值操作符。

14.4.5.2 加减

给原子变量加上某个值：

```
void atomic_add(int i, atomic_t *v);
```

这个函数的功能是将 `v` 指向的原子变量的值加上 `i`。

给原子变量减去某个值：

```
void atomic_sub(int i, atomic_t *v);
```

这个函数的功能是将 `v` 指向的原子变量的值减去 `i`。

使原子变量的值加 1：

```
void atomic_inc(atomic_t *v)
```

这个函数的功能是将 `v` 指向的原子变量的值加上 1。

使原子变量的值减 1：

```
void atomic_dec(atomic_t *v)
```

这个函数的功能是将 `v` 指向的原子变量的值减去 1。

14.4.5.3 加减并返回值

给原子变量加上某个值并返回结果值：

```
int atomic_add_return(int i, atomic_t *v);
```

这个函数的功能是将 `v` 指向的原子变量的值加上 `i`，并将结果返回。
给原子变量减去某个值并返回结果值：

```
int atomic_sub_return(int i, atomic_t *v);
```

这个函数的功能是将 `v` 指向的原子变量的值减去 `i`，并将结果返回。
给原子变量的值加 1 并返回结果值：

```
int atomic_inc_return(atomic_t *v);
```

这个函数的功能是将 `v` 指向的原子变量的值加上 1，并将结果返回。
给原子变量的值减 1 并返回结果值：

```
int atomic_dec_return(atomic_t *v);
```

这个函数的功能是将 `v` 指向的原子变量的值减去 1，并将结果返回。

14.4.5.4 加减并检测

给原子变量加上某个值并检测结果是否为负：

```
int atomic_add_negative(int i, atomic_t *v);
```

这个函数的功能是将 `v` 指向的原子变量的值加上 `i`。返回值非 0 表示结果是负数，为 0 表示结果不是负数。

给原子变量减去某个值并检测结果是否为 0：

```
int atomic_sub_and_test(i, atomic_t *v);
```

这个函数的功能是将 `v` 指向的原子变量的值减去 `i`。返回值非 0 表示结果是 0，为 0 表示结果不是 0。

给原子变量的值加 1 并检测结果是否为 0：

```
int atomic_inc_and_test(atomic_t *v);
```

这个函数的功能是将 `v` 指向的原子变量的值加上 1。返回值非 0 表示结果是 0，为 0 表示结果不是 0。

给原子变量的值减 1 并检测结果是否为 0：

```
int atomic_dec_and_test(atomic_t *v);
```

这个函数的功能是将 `v` 指向的原子变量的值加上 1。返回值非 0 表示结果是 0，为 0 表示结果不是 0。

以上关于原子变量的基本操作可总结如表 14.2 所示。

表 14.2 原子变量的基本操作

操作名	基本操作	有返回值操作	对结果进行检测
加法	<code>atomic_add</code>	<code>atomic_add_return</code>	<code>atomic_add_negative</code>
减法	<code>atomic_sub</code>	<code>atomic_sub_return</code>	<code>atomic_sub_and_test</code>
加 1	<code>atomic_inc</code>	<code>atomic_inc_return</code>	<code>atomic_inc_and_test</code>
减 1	<code>atomic_dec</code>	<code>atomic_dec_return</code>	<code>atomic_dec_and_test</code>

14.4.5.5 条件加法

有条件地给原子变量加上某个值：

```
int atomic_add_unless(atomic_t *v, int a, int u);
```

这个函数的功能如下：如果 `v` 指向的原子变量的原值是 `u` 则无操作并返回 0，否则将它的值加上 `a` 并返回非 0 值。

有条件地给原子变量的值加 1：

```
int atomic_inc_not_zero(atomic_t *v);
```

这个函数的功能如下：如果 `v` 指向的原子变量的原值是 0 则无操作并返回 0，否则将它的值加上 1 并返回非 0 值。

14.4.5.6 值交换

条件交换：

```
int atomic_cmpxchg(atomic_t *v, int old, int new);
```

这个函数的功能如下：如果 `v` 指向的原子变量的值是 `old`，则将其值设为 `new`，否则无操作，返回原子变量的原值。

直接交换：

```
int atomic_xchg(atomic_t *v, int new);
```

这个函数的功能是将 `v` 指向的原子变量的值设为 `new`，并返回它原来的值。

14.4.6 比特位操作

内核中另一类经常用到的操作是对单个比特位进行操作，对此，也有一些相应的接口函数保证操作的原子性。这些操作的实现虽然也是与平台相关的，但可以通过包含以下头文件进行使用：

```
#include <linux/bitops.h>
```

比特位置位：

```
void set_bit(int nr, unsigned long *p);
```

这个函数的功能是将 `p` 指向的长整型数的第 `nr` 个比特位置为 1。

使用这个函数需要注意以下几点。

- ◆ 如果 `nr` 的值超过 31，则所设置的位置将顺延至下一个长整型数中。
- ◆ 无论字节序是大端还是小端，比特位的编号都是从长整型数的最低位开始为 0，到最高位为 31。

下面的其他函数也有类似的特性。

比特位清零：

```
void clear_bit(int nr, unsigned long *p);
```

这个函数的功能是将 `p` 指向的长整型数的第 `nr` 个比特位置为 0。

比特位反转：

```
void change_bit(int nr, unsigned long *p);
```

这个函数的功能是将 `p` 指向的长整型数的第 `nr` 个比特位反转，即原来是 0 则置为 1，原来是 1 则置为 0。

测试并将比特位置位：

```
int test_and_set_bit(int nr, unsigned long *p);
```

这个函数的功能是将 `p` 指向的长整型数的第 `nr` 个比特位置为 1 并返回这个比特位原来的值。

测试并将比特位清零：

```
int test_and_clear_bit(int nr, unsigned long *p);
```

这个函数的功能是将 `p` 指向的长整型数的第 `nr` 个比特位置为 0 并返回这个比特位原来的值。

测试并将比特位反转：

```
int test_and_change_bit(int nr, unsigned long *p);
```

这个函数的功能是将 `p` 指向的长整型数的第 `nr` 个比特位反转，并返回这个比特位原来的值。

14.4.7 互斥体

互斥体及后面要讲的信号灯、读写信号灯都属于非原子操作的同步手段。它们的共同特点是：当获取操作失败需要进行等待时，进程将进入睡眠状态。因为会导致进程睡眠，所以这些同步手段都不能用在原子上下文中。

内核的互斥体与应用编程的线程互斥类似，它有以下特点。

- ◆ 一个互斥体在某一时刻只能被一个进程持有。
- ◆ 互斥体只能由持有它的进程释放。
- ◆ 不能重复获取同一个互斥体。

因为有这些特点，所以它一般用于临界区的保护。内核互斥体的相关定义和声明放在以下头文件中：

```
#include <linux/mutex.h>
```

14.4.7.1 定义与初始化

使用互斥体时首先要定义一个互斥体类型的变量，例如：

```
struct mutex lock; /* 定义一个互斥体变量 lock */
```

然后用以下函数进行初始化：

```
void mutex_init(struct mutex *lock);
```

其中参数 `lock` 指向要初始化的互斥体变量。

也可以直接定义并初始化，如：

```
DEFINE_MUTEX(lock); /* 定义并初始化互斥体变量 lock */
```

对应于初始化操作，还有一个互斥体的销毁操作，其接口函数原型如下：

```
void mutex_destroy(struct mutex *lock);
```

其中参数 `lock` 指向要销毁的互斥体。实际上它被定义为无操作，因此可以省略。

14.4.7.2 获取与释放

获取互斥体的接口函数原型如下：

```
void mutex_lock(struct mutex *lock);
```

这个函数的功能是获取参数 `lock` 指向的互斥体，如果不能获取（互斥体已被其他进程获取），则进程将进入睡眠状态，直到其他进程释放这个互斥体时才被唤醒。函数返回时说明互斥体已被获取。

使用上面这个函数时，进程将进入不可被信号打断的睡眠状态。如果希望进程在等待互斥体时仍然能响应信号，则使用下面的函数：

```
int mutex_lock_interruptible(struct mutex *lock); /* 响应所有信号 */
int mutex_lock_killable(struct mutex *lock); /* 只响应致命信号 */
```

其参数及返回值的含义解释如下。

- ◆ `lock`：指向被获取的互斥体。
- ◆ 返回值：0 表示获取成功，-EINTR 表示在等待互斥体的过程中被信号打断，互斥体没有获取成功。

释放互斥体的接口函数原型如下：

```
void mutex_unlock(struct mutex *lock);
```

其中参数 `lock` 指向要释放的互斥体。

下面这个函数可以尝试获取互斥体，如果不能获取也不会使进程进入睡眠状态，而是反映在返回值上：

```
int mutex_trylock(struct mutex *lock);
```

其参数及返回值的含义解释如下。

- ◆ lock: 指向要尝试获取的互斥体。
- ◆ 返回值: 非 0 表示获取成功, 0 表示获取失败。

注意这个函数虽然不会使进程睡眠, 但由于互斥体只能由获取它的进程释放, 所以不能用在中断上下文中。

14.4.7.3 状态判断

下面这个函数可以判断互斥体是否已被获取:

```
int mutex_is_locked(struct mutex *lock);
```

其参数及返回值的含义解释如下。

- ◆ lock: 指向被判断的互斥体。
- ◆ 返回值: 非 0 表示互斥体已锁, 不能被获取; 0 表示互斥体未锁, 可以被获取。

14.4.8 信号灯

内核也提供了类似于应用编程的信号灯机制。信号灯与互斥体的区别在于有数量的控制, 包含以下几方面的内容。

- ◆ 初始化信号灯时, 可以给它一个正的初值。
- ◆ 进行获取操作时, 如果信号灯的值大于 0, 则获取操作可以成功, 然后它的值将被减去 1, 否则当前进程将进入睡眠状态。
- ◆ 释放操作使信号灯的值加上 1, 如果有进程在等待信号灯, 则将其唤醒。

信号灯的另外一个特点是它的获取者和释放者不必是同一个进程, 因此应用场合比互斥体广泛。事实上, 用初值为 1 的信号灯完全可以模拟互斥体的行为, 但由于互斥体在实现上比信号灯占用的内存更少, 且速度更快, 所以只要情况允许, 就应该尽量使用互斥体。

内核中关于信号灯的定義和声明放在以下头文件中:

```
#include <linux/semaphore.h>
```

14.4.8.1 定义与初始化

使用信号灯前, 首先要定义一个信号灯类型的变量, 如:

```
struct semaphore sem; /* 定义信号灯变量 sem */
```

信号灯在使用前要初始化, 初始化时可以指定信号灯的初值。初始化用以下函数:

```
void sema_init(struct semaphore *sem, int val);
```

这个函数的功能是将参数 sem 指向的信号灯初始化, 初值设为 val。

14.4.8.2 获取与释放

获取信号灯的接口函数原型如下：

```
void down(struct semaphore *sem);
```

其中参数 `sem` 指向要被获取的信号灯。

使用这个函数获取信号灯时，如果信号灯的值已为 0，即信号灯不能被获取，则进程将进入不可被信号打断的睡眠状态。如果希望进程在等待信号灯的过程中仍能响应信号，则要用下面的函数：

```
int down_interruptible(struct semaphore *sem); /* 响应所有信号 */
int down_killable(struct semaphore *sem); /* 只响应致命信号 */
```

其参数及返回值的含义如下。

- ◆ `sem`：指向要获取的信号灯。
- ◆ 返回值：0 表示获取成功，`-EINTR` 表示在等待信号灯的过程中被信号打断，信号灯没有获取成功。

还有一个获取信号灯的接口函数，可以让进程对信号灯的等待操作有时间限制，其原型如下：

```
int down_timeout(struct semaphore *sem, long jiffies);
```

其各个参数及返回值的含义如下。

- ◆ `sem`：指向要获取的信号灯。
- ◆ `jiffies`：等待时间的上限。
- ◆ 返回值：0 表示获取成功，`-ETIME` 表示在等待信号灯的过程中超时，信号灯获取失败。

释放信号灯的接口函数原型如下：

```
void up(struct semaphore *sem);
```

其中参数 `sem` 指向被释放的信号灯。

对于信号灯的获取操作，也有一个“尝试”的版本，其原型如下：

```
int down_trylock(struct semaphore *sem);
```

其参数及返回值的含义解释如下。

- ◆ `sem`：指向要尝试获取的信号灯。
- ◆ 返回值：0 表示获取成功，非 0 表示获取失败。



尝试获取信号灯函数的返回值含义与互斥体、自旋锁等的对应函数完全相反。

与用于互斥体的 `mutex_trylock` 函数不同，这个函数可以安全地用在中断上下文中。

14.4.9 读写信号灯

读写信号灯虽然名称上有“信号灯”，但它更像是互斥体的“读写”版本。读写信号灯与互斥体的关系可以跟读写锁与自旋锁的关系类比。与互斥体一样，读写信号灯的获取与释放必须由同一个进程操作，并且不能重复获取。

读写信号灯的获取操作被分为两类：读信号灯获取和写信号灯获取。与读写锁的概念类似，它允许多个进程同时获取读信号灯，但只能有一个进程获取写信号灯，且写信号灯与读信号灯互相排斥。

内核中关于读写信号灯的接口放在以下头文件中：

```
#include <linux/rwsem.h>
```

14.4.9.1 定义和初始化

使用读写信号灯前首先要定义一个读写信号灯类型的变量，如：

```
struct rw_semaphore sem; /* 定义读写信号灯类型的变量 sem */
```

用以下函数将读写信号灯变量初始化：

```
void init_rwsem(struct rw_semaphore *sem);
```

其中参数 `sem` 指向要初始化的读写信号灯。

也可以直接定义并初始化，如：

```
DECLARE_RWSEM(sem); /* 定义并初始化读写信号灯变量 sem */
```

14.4.9.2 获取与释放

获取读信号灯的接口函数原型如下：

```
void down_read(struct rw_semaphore *sem);
```

其中参数 `sem` 指向要获取的读写信号灯。

获取写信号灯的接口函数原型如下：

```
void down_write(struct rw_semaphore *sem);
```

其中参数 `sem` 指向要获取的读写信号灯。

释放读信号灯的接口函数原型如下：

```
void up_read(struct rw_semaphore *sem);
```

其中参数 `sem` 指向要释放的读写信号灯。

释放写信号灯的接口函数原型如下：

```
void up_write(struct rw_semaphore *sem);
```

其中参数 `sem` 指向要释放的读写信号灯。

以下是获取读写信号灯操作的“尝试”版本：

```
int down_read_trylock(struct rw_semaphore *sem); /* 尝试获取读信号灯 */
int down_write_trylock(struct rw_semaphore *sem); /* 尝试获取写信号灯 */
```

其中参数 `sem` 指向要尝试获取的读写信号灯，返回值非 0 表示获取成功，为 0 表示获取失败，这一点与信号灯的尝试获取操作是相反的。

读写信号灯有一个特殊的操作为信号灯的降格。当一个进程获取到写信号灯并进入临界区后，可能只做少量的写操作，随后全部是只读操作，这时，可以在写操作完成后将信号灯降格为读信号灯，这样就可以使其他等待读信号灯的进程继续执行，从而提高系统的运行效率。使读写信号灯降格的接口函数原型如下：

```
void downgrade_write(struct rw_semaphore *sem);
```

其中参数 `sem` 指向要降格的读写信号灯。

这个函数并不等同于先释放写信号灯，再获取读信号灯，它不会导致进程睡眠。如果先释放写信号灯，则再获取读信号灯时可能已有其他进程获取了写信号灯，从而导致进程进入睡眠状态。

14.4.9.3 状态判断

内核还提供了—个接口函数用于判断读写信号灯是否已被获取：

```
int rwsem_is_locked(struct rw_semaphore *sem);
```

其参数及返回值的含义解释如下。

- ◆ `sem`：指向被判断的读写信号灯。
- ◆ 返回值：非 0 表示已被获取（无论是读信号灯还是写信号灯），0 表示未被获取。

14.5 端口 IO 和内存映射 IO

设备驱动程序实质上可以视为内核功能的补充。一方面，它向内核提供各种操作函数，使内核能够正确地管理某个特定的硬件设备；另一方面，在这些函数里，不可避免地要与硬件设备通信，包括输入数据和输出数据两个方面。输入和输出一般利用内核提供的底层操作接口来完成，在极个别情况下，驱动也会直接用指令访问硬件设备。

驱动向内核提供的函数可分为两类：一类用于支持内核的系统调用，如字符设备的打开、关闭、读和写等操作，另一类用于支持设备中断。应用程序、内核、驱动的层次关系如图 14.5 所示。

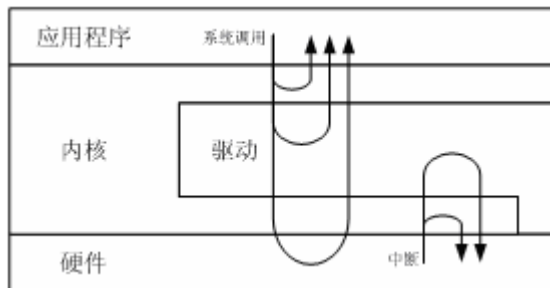


图 14.5 软件层次关系

计算机系统的外部设备可分为两类：一类没有具体的功能，它的唯一作用就是用来连接其他设备，如系统的 PCI 总线控制器、USB 总线控制器等；另外一类设备则有具体的功能，如键盘、鼠标、显示设备等。设备之间形成树状的连接关系，只有处在树叶位置的设备才有具体的功能。

处于树根位置的设备由 CPU 直接控制，软件上直接通过 CPU 指令来操作，这样的设备往往是总线控制器，没有具体的功能，在它的驱动里，一般只是提供一些内核的编程接口，供那些连接在其上的设备驱动使用；其他的设备不能直接由 CPU 控制，它的驱动在软件上位于总线控制器驱动的上层，使用后者提供的编程接口来操控设备。这样，对应于硬件上的设备连接关系，软件上也形成了驱动栈的层次关系，如图 14.6 所示，这里举了一个 USB 设备驱动的例子。

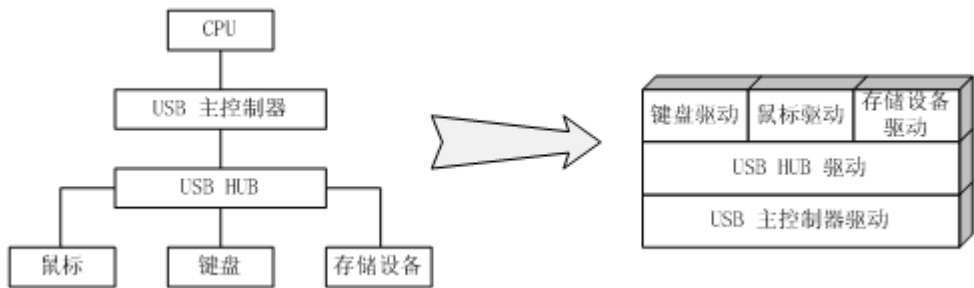


图 14.6 设备连接关系与驱动栈

设备与 CPU 之间的数据交换有两种基本方式：端口方式和内存映射方式。

端口方式需要 CPU 有特殊的地址、数据及控制总线与设备连接，并且通过专门的 IO 指令来操作。这时，端口的地址空间与内存的物理地址空间是相互独立的，甚至它们的位数都可以不一样。端口的地址一般称为端口号。

在内存映射方式下，CPU 将把设备当做内存来处理。这时要求设备接入内存总线并且具有与内存相同的电信号接口。一般来说，由于需要内存管理和控制单元，访问内存的电路系统要比访问端口复杂，但由于 CPU 天然支持内存访问，因此内存映射方式反而不需要增加额外电路。它的缺点是要占用内存的物理地址空间。

多数 CPU 架构都只支持内存映射方式，包括 ARM 架构；部分 CPU 架构既支持内存映射方式又支持端口方式，如 x86 架构。对于不支持端口方式的 CPU，通过一个端口控制器的转接，也可以连接那些需要端口操作的设备。Linux 内核对两种方式都提供了相应的底层支持。

以上两种方式都是 CPU 直接访问设备的方式。还有一种直接内存访问（DMA）的方式可以让外部设备直接与内存交换数据。这时需要 DMA 控制器的支持，它可以不经过 CPU 而直接读写内存。CPU 把需要读写的内存区域的地址和大小告知 DMA 控制器，然后下达开始指令，DMA 控制器就可以自动完成读写操作，操作完成后通过中断的方式通知 CPU。DMA 方式能够大大提高设备输入输出的速度。需要注意的是，DMA 控制器不一定能访问全部系统内存，并且它访问内存所用的地址也不一定与 CPU 访问内存所用的物理地址相同。

14.5.1 端口 IO

14.5.1.1 端口的申请和释放

内核将端口号视为一种资源而加以管理，驱动在使用端口前必须先进行申请。这样可以防止多

个驱动同时访问一个端口而造成混乱。申请和释放端口需要包含以下头文件：

```
#include <linux/ioport.h>
```

申请使用端口号的接口函数原型如下：

```
struct resource *request_region(resource_size_t start, resource_size_t n,
    const char *name);
```

其各个参数及返回值的含义解释如下。

- ◆ start：要申请的端口号的起始值。
- ◆ n：要申请的端口号的连续个数。
- ◆ name：赋给申请到的资源的名称，一般就用设备名。
- ◆ 返回值：指向申请到的资源，NULL 表示申请失败。

这个函数可以申请一段端口号的区域，其返回值指向内核用于管理资源的一个结构体，一般只看这个指针是不是 NULL 而不关心结构体的具体内容。端口号的分配情况可查看 `/proc/ioports` 文件，其中出现的设备名就是申请端口时提供的 `name` 参数。

参数的 `resource_size_t` 类型实际定义如下：

```
typedef unsigned int u32;
typedef u32 phys_addr_t;
typedef phys_addr_t resource_size_t;
```

可见它实际上就是无符号整型。

端口资源在停止使用后要释放，其接口函数原型如下：

```
void release_region(resource_size_t start, resource_size_t n);
```

其各个参数的含义解释如下。

- ◆ start：要释放的端口号的起始值。
- ◆ n：要释放的端口号的连续个数。

端口号的释放操作必须和申请操作一一对应。

14.5.1.2 端口的读写

端口申请成功以后就可以进行读写操作了，读写端口的函数都声明在以下头文件中：

```
#include <linux/io.h>
```



提示

读写端口的接口函数与申请和释放端口的接口函数不在同一个头文件中。

读一个 8 比特数据（字节）：

```
unsigned char inb(unsigned long port);
```

读一个 16 比特数据（字）：

```
unsigned short inw(unsigned long port);
```

读一个 32 比特数据（双字）：

```
unsigned int inl(unsigned long port);
```

这三个函数的参数 `port` 是要读的端口号，返回值是读到的数据。端口号的类型是与平台相关的，有些平台是 16 位的，有些平台是 32 位的。

写一个 8 比特数据：

```
void outb(unsigned char v, unsigned long port);
```

写一个 16 比特数据：

```
void outw(unsigned short v, unsigned long port);
```

写一个 32 比特数据：

```
void outl(unsigned int v, unsigned long port);
```

这三个函数的参数 `v` 是要写的数据，参数 `port` 是端口号。

14.5.1.3 端口的串操作

有些 CPU 支持对端口的串操作指令，当向一个端口连续输出数据时，可以利用这些指令提高效率。内核将这些指令包装成了相应的接口函数，在 CPU 不支持串操作指令的情况下，内核将使用循环来实现这些函数。

串读多个 8 比特数据：

```
void insb(unsigned long port, void *buf, int n);
```

串读多个 16 比特数据：

```
void insw(unsigned long port, void *buf, int n);
```

串读多个 32 比特数据：

```
void insl(unsigned long port, void *buf, int n);
```

这三个函数的各个参数的含义解释如下。

- ◆ `port`：要读的端口号。
- ◆ `buf`：指向一个内存缓冲区，用来存放读到的数据。
- ◆ `n`：读取的数据个数，注意不是字节数。

串写多个 8 比特数据：

```
void outsb(unsigned long port, const void *buf, int n);
```

串写多个 16 比特数据:

```
void outsw(unsigned long port, const void *buf, int n);
```

串写多个 32 比特数据:

```
void outsl(unsigned long port, const void *buf, int n);
```

这三个函数的各个参数的含义解释如下。

- ◆ port: 要写的端口号。
- ◆ buf: 指向一个内存缓冲区, 要写的数据逐个从这里取出。
- ◆ n: 要写的数据个数, 注意不是字节数。

串操作与一般的内存复制操作不同, 它对端口的读写操作都发生在同一端口号上, 但对内存的读写操作是逐次后移的。另外, 给串操作函数提供的数据个数并不是字节数, 而是所读写的数据的个数。例如用以下的代码从端口串读数据:

```
insl(port, buf, 8); /* 从端口 port 读 8 个双字放在 buf 指向的缓冲区中 */
```

由于所读的数据是 4 个字节的, 所以用来放置这些数据的缓冲区的长度至少应是 32 字节。对于端口的读写操作函数可总结如表 14.3 所示。

表 14.3 端口的读写操作

操作	8 比特	16 比特	32 比特
读	inb	inw	inl
写	outb	outw	outl
串读	insb	insw	insl
串写	outsb	outsw	outsl

14.5.2 内存映射 IO

14.5.2.1 IO 内存的申请与释放

内核用相同的机制来管理端口资源和 IO 内存资源, 因此它们的接口函数放在相同的头文件中:

```
#include <linux/ioport.h>
```

申请 IO 内存的接口函数原型如下:

```
struct resource *request_mem_region(resource_size_t start, resource_size_t n,
    const char *name);
```

其各个参数及返回值的含义解释如下。

- ◆ start: 要申请的 IO 内存的起始地址。
- ◆ n: 要申请的 IO 内存的大小, 单位是字节。

- ◆ **name**: 赋给申请到的资源的名称, 一般就用设备名。
- ◆ **返回值**: 指向申请到的资源, **NULL** 表示申请失败。

这个函数可以申请一段 IO 内存区域。注意申请时使用的内存地址是物理地址, 故用整型来表示, 而不是指针型。IO 内存的分配情况可查看 `/proc/iomem` 文件, 其中出现的设备名就是申请 IO 内存时提供的 **name** 参数。

IO 内存存在停止使用后要释放, 其接口函数原型如下:

```
void release_mem_region(resource_size_t start, resource_size_t n);
```

其各个参数的含义解释如下。

- ◆ **start**: 要释放的 IO 内存的起始地址。
- ◆ **n**: 要释放的 IO 内存的字节数。

IO 内存的释放操作必须和申请操作一一对应。

14.5.2.2 IO 内存的映射

与端口的操作不同, 申请时用的 IO 内存地址是物理地址, 并不能直接访问。要想操作 IO 内存, 还必须将其物理地址映射成内核的虚拟地址。IO 内存的映射及读写操作都放在以下头文件中:

```
#include <linux/io.h>
```

映射操作的接口函数原型如下:

```
void __iomem *ioremap(unsigned long phys_addr, size_t size);
```

其各个参数及返回值的原型如下。

- ◆ **phys_addr**: 被映射的 IO 内存的起始物理地址。
- ◆ **size**: 被映射的 IO 内存区域的长度。
- ◆ **返回值**: 映射后的虚拟地址, **NULL** 表示映射失败。

映射成功后得到的地址是内核的虚拟地址, 可以直接访问, 但不推荐直接用赋值操作符, 最好用内核提供的读写函数。`__iomem` 用来表示它是用于 IO 的内存地址, 以引起编程者的注意, 无任何实际作用。

映射好的 IO 内存存在不用时需要解除映射, 接口函数如下:

```
void iounmap(void __iomem *io_addr);
```

其中参数 **io_addr** 是映射得到的虚拟地址。

14.5.2.3 IO 内存的读写

对 IO 内存进行读写时, 使用的都是映射后的虚拟地址。
从 IO 内存读一个 8 比特数据:


```
unsigned int ioread8(const void __iomem *addr);
```

从 IO 内存读一个 16 比特数据:

```
unsigned int ioread16(const void __iomem *addr);
```

从 IO 内存读一个 32 比特数据:

```
unsigned int ioread32(const void __iomem *addr);
```

这三个函数的参数和返回值的含义解释如下。

- ◆ addr: 要读的 IO 内存的虚拟地址。
- ◆ 返回值: 读到的数据。

注意虽然读取的数据有各种类型,但返回值都是无符号整型,使用时强制转换为所需的类型即可。

向 IO 内存写入一个 8 比特数据:

```
void iowrite8(unsigned char v, void __iomem *addr);
```

向 IO 内存写入一个 16 比特数据:

```
void iowrite16(unsigned short v, void __iomem *addr);
```

向 IO 内存写入一个 32 比特数据:

```
void iowrite32(unsigned int v, void __iomem *addr);
```

这三个函数的各个参数的含义解释如下。

- ◆ v: 要写入的数据。
- ◆ addr: 要写的 IO 内存的虚拟地址。

14.5.2.4 IO 内存的串操作

对于一般的内存,连续对一个内存地址读写多个数据是没有意义的,但是对于设备的 IO 操作,连续输入或输出多个数据是一种常见的操作。与端口 IO 类似,内核也提供了相应的串操作函数。

串读多个 8 比特数据:

```
void ioread8_rep(void __iomem *addr, void *buf, int n);
```

串读多个 16 比特数据:

```
void ioread16_rep(void __iomem *addr, void *buf, int n);
```

串读多个 32 比特数据:

```
void ioread32_rep(void __iomem *addr, void *buf, int n);
```

这三个函数的各个参数的含义解释如下。

- ◆ addr: 要读的 IO 内存的虚拟地址。
- ◆ buf: 指向一个内核缓冲区, 用来存放读到的数据。
- ◆ n: 要读的数据个数, 注意不是字节数。

串写多个 8 比特数据:

```
void iowrite8_rep(void __iomem *addr, const void *buf, int n);
```

串写多个 16 比特数据:

```
void iowrite16_rep(void __iomem *addr, const void *buf, int n);
```

串写多个 32 比特数据:

```
void iowrite32_rep(void __iomem *addr, const void *buf, int n);
```

这三个函数的各个参数的含义解释如下。

- ◆ addr: 要写的 IO 内存的虚拟地址。
- ◆ buf: 指向一个内核缓冲区, 要写入的数据事先存放在这里。
- ◆ n: 要写的数据个数, 注意不是字节数。

与端口 IO 的串操作类似, 这里的串操作对 IO 内存的读写也是发生在同一内存地址上的, 并且也要注意给串操作函数提供的数据个数并不是字节数, 而是所读写的数据的个数。

以上对于 IO 内存的操作可总结如表 14.4 所示。

表 14.4 IO 内存的读写操作

操作	8 比特	16 比特	32 比特
读	ioread8	ioread16	ioread32
写	iowrite8	iowrite16	iowrite32
串读	ioread8_rep	ioread16_rep	ioread32_rep
串写	iowrite8_rep	iowrite16_rep	iowrite32_rep

14.5.2.5 IO 内存的旧操作接口

早期版本的内核有另外一些用于操作 IO 内存的接口, 由于它们在驱动中大量使用, 因此在 Linux 2.6.30 内核中仍然保留了这些接口, 但是不推荐使用。这些接口列举如下:

```
unsigned char readb(void __iomem *addr); /* 读 8 比特数据 */
unsigned short readw(void __iomem *addr); /* 读 16 比特数据 */
unsigned int readl(void __iomem *addr); /* 读 32 比特数据 */
void writeb(unsigned char v, void __iomem *addr); /* 写 8 比特数据 */
void writew(unsigned short v, void __iomem *addr); /* 写 16 比特数据 */
void writel(unsigned int v, void __iomem *addr); /* 写 32 比特数据 */
void readsb(void __iomem *addr, void *buf, int n); /* 串读 8 比特数据 */
void readsw(void __iomem *addr, void *buf, int n); /* 串读 16 比特数据 */
void readsl(void __iomem *addr, void *buf, int n); /* 串读 32 比特数据 */
```

```
void writesb(void __iomem *addr, void *buf, int n); /* 串写 8 比特数据 */
void writesw(void __iomem *addr, void *buf, int n); /* 串写 16 比特数据 */
void writel(void __iomem *addr, void *buf, int n); /* 串写 32 比特数据 */
```

14.5.2.6 IO 内存的内存操作

与端口操作不同，IO 内存有可能对应着设备上一块真正的内存，因此，对 IO 内存实现类似于普通内存的一些操作是有意义的。内核中为此提供了一些接口函数。

设置 IO 内存的接口函数原型如下：

```
void memset_io(void __iomem *addr, int c, size_t count);
```

其各个参数的含义解释如下。

- ◆ addr: 要设置的 IO 内存的起始地址。
- ◆ c: 要设置的值。
- ◆ count: 要设置的 IO 内存区域的长度，单位是字节。

这个函数的功能类似于 `memset` 函数，只不过操作的是 IO 内存。

从 IO 内存复制数据到内核内存的接口函数原型如下：

```
void memcpy_fromio(void *dst, const void __iomem *src, size_t count);
```

从内核内存复制数据到 IO 内存的接口函数原型如下：

```
void memcpy_toio(void __iomem *dst, const void *src, size_t count);
```

这两个函数的各个参数的含义解释如下。

- ◆ dst: 指向目标内存区域。
- ◆ src: 指向源内存区域。
- ◆ count: 要复制的内存区域的大小，单位是字节。

类似于 `memcpy` 函数，这两个函数进行操作时，数据的读写位置是随着复制的过程而不断后移的。

14.5.2.7 将端口映射为 IO 内存

内核还提供了将端口映射为 IO 内存的接口函数，映射以后就可以用读写 IO 内存的函数来读写相应的端口，其原型如下：

```
void __iomem *ioproport_map(unsigned long port, unsigned int n);
```

其各个参数及返回值的含义解释如下。

- ◆ port: 要映射的起始端口号。
- ◆ n: 要映射的端口号的连续个数。
- ◆ 返回值: 映射后的 IO 内存的虚拟地址。

对应的取消映射的接口函数原型如下：

```
void ioport_unmap(void __iomem *addr);
```

其中 `addr` 是原来映射时得到的 IO 内存虚拟地址。

14.5.3 内存屏障

IO 内存与普通内存有不同的行为，而编译器通常针对普通内存的读写操作对代码进行优化，例如下面的代码：

```
*addr = 1; /* 向 addr 指向的内存单元写入 1 */
*addr = 2; /* 向 addr 指向的内存单元写入 2 */
*addr = 3; /* 向 addr 指向的内存单元写入 3 */
```

经编译器优化之后，实际上只有最后一次写操作生成了代码，其他的代码全部被认为是多余的。如果 `addr` 指向的是 IO 内存，那么这段代码实际上没有达到预期的效果。对于这种情况，可以使用 `volatile` 关键字来定义变量，如：

```
volatile char *addr;
```

这个定义告诉编译器，`addr` 指向的变量是一种受外界影响、不经过写操作就会发生变化的变量。对这种变量进行操作时，编译器必须忠实地翻译每个读写操作，不能做优化。

但是这还不足以解决所有优化造成的问题，比如编译器会根据 CPU 的特点调整指令顺序，有可能会对设备的 IO 操作造成影响。另外，CPU 在执行指令时，为了提高流水线的效率，也有可能对执行顺序做一些调整。再考虑 CPU 缓存的作用，即使代码不经优化，所做的读写操作也不一定真正发生在内存里。为了解决这些问题，可以使用内核提供的内存屏障接口，通过向代码中插入一些语法元素和汇编指令，防止编译器对代码进行优化，以及 CPU 对指令执行的优化，并且保证所做的读写操作可靠地发生在内存中。

防止编译器优化可以用内核的软屏障接口函数，原型如下：

```
void barrier(void);
```

这个函数可以阻止编译器进行跨越它的优化，例如将上面的代码改造成如下代码：

```
*addr = 1; /* 向 addr 指向的内存单元写入 1 */
barrier();
*addr = 2; /* 向 addr 指向的内存单元写入 2 */
barrier();
*addr = 3; /* 向 addr 指向的内存单元写入 3 */
barrier();
```

这样，每个写内存的操作都会如实地生成代码。

如果要阻止 CPU 的优化及缓存的问题，则使用硬屏障，它们的原型如下：

```
void rmb(void); /* 读屏障 */
void wmb(void); /* 写屏障 */
```

```
void mb(void); /* 读写屏障 */
```

其中 `rmb` 函数能够保证它之前的读内存操作一定真实地发生在它之后的读内存操作之前；`wmb` 函数能够保证它之前的写内存操作一定真实地发生在它之后的写内存操作之前；`mb` 函数的功能是两者的综合，能够保证它之前的读写内存操作一定真实地发生在它之后的读写内存操作之前。

需要注意的是，这些函数虽然本身不做任何有用操作，但由于阻止了 CPU 的优化，也会导致执行效率的降低，故不应该滥用。

很多时候，硬屏障只对多处理器平台有必要，这时可以用以下几个函数：

```
void smp_rmb(void); /* 读屏障 */
void smp_wmb(void); /* 写屏障 */
void smp_mb(void); /* 读写屏障 */
```

如果编译成 SMP 版本，这几个函数与对应的没有 `smp_` 前缀的硬屏障函数是相同的；如果编译成 UP 版本，它们全部转化为 `barrier`。

14.6 中断

外部设备的工作一般不可能与 CPU 同步，例如，当需要向设备输出信息时，设备却处于繁忙状态不能接受数据。访问设备的时机可由以下两种方式确定。

- ◆ 轮询方式：软件上周期性查询设备的状态，看是否可以开始输入输出，如果可以则开始读写数据。
- ◆ 中断方式：当设备的状态发生改变时，通过专门的中断线主动向 CPU 发出电信号，CPU 开始执行中断处理程序，在中断处理中读写数据。

轮询方式需要 CPU 周期性地执行查询设备的操作，如果查询周期太长，则降低设备的响应速度，如果查询周期太短，则消耗大量 CPU 资源。中断方式克服了这个缺点，既能保证设备的请求得到快速响应，又不浪费 CPU 资源。但是，当设备的速度非常高时，中断发生得过于频繁，本身成为巨大的开销，这时由于设备几乎总是处于可以输入输出的状态，轮询方式反而更有效率。

由于中断对于各种 CPU 有普遍意义，内核接管了中断设置的硬件接口（如中断向量表等），并包装成统一的中断申请释放和中断处理函数接口。使用这些接口时需要包含以下头文件：

```
#include <linux/interrupt.h>
```

14.6.1 申请和释放中断

中断在使用前必须先进行申请，其接口函数原型如下：

```
int request_irq(unsigned int irq, irq_handler_t handler,
               unsigned long flags, const char *name, void *dev);
```

其各个参数及返回值的含义解释如下。

- ◆ **irq**: 要申请的中断号。
- ◆ **handler**: 所提供的中断处理函数。
- ◆ **flags**: 申请中断的标志。
- ◆ **name**: 设备的名称。
- ◆ **dev**: 设备标识, 一个指针值, 用来区别共享中断的各个设备。
- ◆ **返回值**: 0 表示成功, 负数为错误码。

申请中断的标志有以下常用取值。

- ◆ **IRQF_DISABLED**: 表示内核要在调用中断处理函数前自动禁止此中断, 调用完毕后自动使能此中断。
- ◆ **IRQF_SAMPLE_RANDOM**: 表示中断可以对系统的熵池产生贡献, 熵池与系统的随机数产生有关系。
- ◆ **IRQF_SHARED**: 表示要与其他设备共享同一个中断号。
- ◆ **IRQF_TRIGGER_RISING**: 表示中断为上升沿触发方式。
- ◆ **IRQF_TRIGGER_FALLING**: 表示中断为下降沿触发方式。
- ◆ **IRQF_TRIGGER_HIGH**: 表示中断为高电平触发方式。
- ◆ **IRQF_TRIGGER_LOW**: 表示中断为低电平触发方式。

这些标志可以用“按位或”操作符组合起来使用。需要注意的是, 并不是任何中断都支持设置触发方式。

中断的使用情况可以通过 `/proc/interrupts` 文件查询, 其中的设备名称就是申请中断时提供的 **name** 参数。

设备标识对内核来说并没有特殊含义, 如果多个设备共享中断, 则只要它们申请时提供的设备标识不同即可; 如果设备独占中断, 则这个参数可以传值为 **NULL**。设备标识的值会原封不动地传入中断处理函数, 因此一般情况下会将表示设备数据的地址传进去以方便在中断处理函数内使用。

申请中断的同时就注册了中断处理函数。申请成功后, 如果发生对应的中断, **CPU** 先执行内核的中断前期处理, 然后调用注册的中断处理函数。

与中断的申请操作对应的是释放操作, 其接口函数原型如下:

```
void free_irq(unsigned int irq, void *dev);
```

其各个参数的含义解释如下。

- ◆ **irq**: 要释放的中断号。
- ◆ **dev**: 设备标识, 应与申请中断时所提供设备标识相同。

释放中断后, 内核就不会再调用申请时提供的中断处理函数。

14.6.2 中断处理函数

中断处理函数的指针类型定义如下:

```
typedef irqreturn_t (*irq_handler_t)(int irq, void *dev);
```

其各个参数和返回值的含义解释如下。

- ◆ irq: 中断号。
- ◆ dev: 等于申请中断时提供的设备标识。
- ◆ 返回值: 表示中断的处理情况。

中断处理函数由驱动向内核提供, 当发生中断时, 内核将回调所注册的中断处理函数, 同时将发生的中断号传入。除非内核本身有缺陷, 否则传入的中断号一定等于申请中断时指定的中断号, 因此这个参数几乎没有什么作用。

内核同时会将设备标识的值传入中断处理函数, 因此可以利用这个参数在申请中断时向中断处理函数传递数据。

中断处理函数的返回值类型定义如下:

```
enum irqreturn {IRQ_NONE, IRQ_HANDLED, IRQ_WAKE_THREAD, };

typedef enum irqreturn irqreturn_t;
```

一般情况下都返回 `IRQ_HANDLED`。

中断处理函数在中断上下文中执行, 它在实现上有一个原则就是要尽快地执行完毕返回。因为中断时 CPU 处在特殊的状态, 并且往往会禁止一些中断的发生, 如果一个中断的处理时间过长, 则会严重影响整个系统的响应速度, 甚至导致系统崩溃。

如果中断发生后有比较耗时的工作要处理, 则应该考虑把这些工作延期执行。这时, 中断处理函数本身称为中断处理的上半部 (`top half`), 被延期执行的部分称为中断处理的下半部 (`bottom half`)。中断处理的上半部执行那些必须立刻完成的操作, 例如从设备读取数据到内核缓冲区; 中断处理的下半部执行时间要求不严格的工作, 例如对数据进行分析处理等。根据需要, 延期工作可以用 `tasklet` 或工作队列来实现。

初看起来, 将中断分为两个部分执行似乎并没有减轻 CPU 的负担, 反而增加了延期工作的开销。实际上, 上半部在每次发生中断时执行, 但执行一次下半部却可以对多次中断的结果一起进行处理, 有可能提高效率。

14.6.3 中断的禁止和使能

内核提供了禁止某个中断号的接口函数, 原型如下:

```
void disable_irq(unsigned int irq);
```

其中参数 `irq` 是要禁止的中断号。

这个函数在禁止中断时, 如果对应的中断处理函数正在执行, 则会一直等待它执行完毕再返回, 这种等待是通过一个空循环来实现的, 不会导致进程睡眠。如果不需要这种等待操作, 则可以用下面这个函数:

```
void disable_irq_nosync(unsigned int irq);
```

其中参数 `irq` 是要禁止的中断号。



不要在中断的处理函数内用 `disable_irq` 函数禁止自身，这样将会立刻死锁，可以使用 `disable_irq_nosync` 函数。

被禁止的中断还可以再使能，接口函数如下：

```
void enable_irq(unsigned int irq);
```

其中参数 `irq` 是要使能的中断号。

禁止和使能的操作次数是有记录的，也就是说，如果一个中断号从最初的使能状态开始被禁止两次，则必须再使能两次才能回到使能状态。

由于申请中断时可以使用 `IRQF_DISABLED` 标志，所以一般不需要在中断处理函数中使用这些函数来禁止或使能中断。一种应用场合是设备时而以中断方式工作，时而以轮询方式工作，这时可以在设备切换到轮询方式时禁止相应的中断以节约 CPU 资源，切换回中断方式时再使能相应的中断。

14.6.4 线程化中断

Linux 2.6.30 内核支持线程化中断的概念。线程化的中断用以下函数进行申请：

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
                        irq_handler_t thread_fn, unsigned long flags,
                        const char *name, void *dev_id);
```

与 `request_irq` 函数相比，它多了一个参数 `thread_fn`，这个参数的类型与中断处理函数相同。实际上，`request_irq` 函数就是对这个函数的包装，只不过 `thread_fn` 参数传递为 `NULL` 而已。

用这个函数申请中断时，如 `thread_fn` 参数的值不是 `NULL`，就会为这个中断号建立一个新的内核线程，名称为 `irq/%d-%s`，其中 `%d` 替换为中断号，`%s` 替换为设备名称。这个线程专门负责执行函数 `thread_fn`。

发生中断时，中断处理函数 `handler` 仍然先被执行。如果它返回 `IRQ_WAKE_THREAD`，就会将对应的中断线程唤醒以执行函数 `thread_fn`。因此线程化的中断一定程度上类似于中断的下半部机制。

14.6.5 共享中断

与内存地址、端口号不同，中断号在系统中属于稀缺资源，一般只有几十到上百个，因此有时必须多个设备共享一个中断号。

共享中断时，驱动仍然用 `request_irq` 或 `request_threaded_irq` 函数进行申请，只是必须有 `IRQF_SHARED` 标志，否则不能与其他驱动共享中断。这时，每个申请者传入的设备标识必须各不相同，因为在释放中断时，内核必须根据设备标识来确定是哪一个申请者要求释放。

内核是这样来处理共享中断的：每一个中断号有一个对应的链表，用来存放所有已注册的中断

处理函数，当中断发生时，内核将逐个调用链表中的处理函数。

这样就有可能出现如下情况：实际上设备并没有产生中断，但驱动为它注册的中断处理函数却被内核调用了。因此，在共享中断时，驱动必须在中断处理函数中检测设备是否真正发生中断，如果没有，则立刻返回 `IRQ_NONE`；如果真的发生了中断，则进行处理，最后返回 `IRQ_HANDLED` 或 `IRQ_WAKE_THREAD`。



第 15 章 Linux 2.6 设备模型

由于各种热插拔设备特别是 USB 设备的大量使用，内核必须处理在系统运行时设备的动态插入和拔除，以及设备之间复杂的依赖关系。为此，Linux 2.6 内核提出了全新的设备模型。新设备模型的核心概念是内核对象及内核集合，从此出发，应用面向对象的思想扩展出了许多新的数据类型，如设备、总线等，以对各种外部设备进行有效的管理。

新的设备模型仅用于设备的管理，它与 Linux 传统的字符设备、块设备等接口并不冲突。驱动仍然可以直接注册字符设备和块设备等功能设备，如果希望设备纳入新的管理体系之下，则还要向内核注册新的设备对象。

15.1 对象与集合

这一节将讲述内核设备模型中最核心的数据类型：内核对象与内核集合。虽然在驱动编程中一般不会直接使用它们，但是对这些核心机制的了解非常有助于理解新的设备模型的编程思想，从而理解各种具体驱动的编程接口。

15.1.1 引用计数

新设备模型的一个突出特点就是各种数据类型都支持引用计数。以动态创建的数据对象为例，当需要引用它时，就使它的引用计数加 1；当不需要它时，就使它的引用计数减 1；只有当引用计数变为 0 时，这个数据对象才被销毁。这样就避免了将其他模块正在使用的数据对象销毁的现象。设备模型中的所有数据类型都在使用下面介绍的引用计数机制。

15.1.1.1 引用计数类型

表示引用计数的数据类型定义如下：

```
struct kref {
    atomic_t refcount; /* 用于计数的原子型变量 */
};
```

它只有一个成员用来记录引用计数。

15.1.1.2 引用计数基本操作

设置引用计数的值：

```
void kref_set(struct kref *kref, int num);
```

其中参数 kref 指向要设置的引用计数变量，参数 num 是要设置的值。

初始化引用计数的值（设置为 1）：

```
void kref_init(struct kref *kref);
```

其中参数 `kref` 指向要初始化的引用计数变量。

获取引用计数（使它的值加 1）：

```
void kref_get(struct kref *kref);
```

其中参数 `kref` 指向要获取的引用计数变量。

释放引用计数（使它的值减 1）：

```
int kref_put(struct kref *kref, void (*release)(struct kref *kref));
```

其各个参数及返回值的含义解释如下。

- ◆ `kref`：指向要释放的引用计数变量。
- ◆ `release`：是一个函数指针，当引用计数的值降为 0 时，这个函数会被调用。
- ◆ 返回值：1 表示引用计数的值已降为 0，0 表示没有降为 0。

`release` 指向的函数可称为释放函数。一般来说，支持引用计数的数据类型会嵌套一个 `struct kref` 型成员，并提供一个释放函数。

15.1.2 内核对象

内核对象是设备模型中最基本的数据类型，它有名称，支持引用计数，可以有父子关系，并且可以属于某个内核集合。内核对象与 `sysfs` 文件系统中的目录一一对应，其父子关系对应着目录的层次关系，其属性对应着目录中的文件。

15.1.2.1 内核对象数据类型

表示内核对象的数据类型定义如下：

```
struct kobject {
    const char          *name; /* 名称 */
    struct list_head    entry; /* 链表节点 */
    struct kobject      *parent; /* 指向父对象 */
    struct kset          *kset; /* 指向所属内核集合 */
    struct kobj_type     *ktype; /* 指向一个描述内核对象类型的结构体 */
    struct kref          kref; /* 引用计数 */
    unsigned int        uevent_suppress:1; /* 表示用户态事件是否被抑制 */
    /* 这里只写出了部分成员 */
};
```

其各个成员的含义解释如下。

- ◆ `name`：指向一个字符串，用来保存内核对象的名称。
- ◆ `entry`：链表节点，用于把属于同一个集合的多个内核对象组成链表。
- ◆ `parent`：用于维护内核对象的父子关系。

- ◆ **kset**: 指向所属的内核集合。
- ◆ **ktype**: 指向描述内核对象类型的结构体, 包括对象的属性和销毁方法等。
- ◆ **kref**: 用于实现内核对象的引用计数。
- ◆ **uevent_suppress**: 一个标志位, 用于表示用户态事件是否被抑制。

15.1.2.2 内核对象的名称

内核对象有名称属性, 可以通过以下函数设置:

```
int kobject_set_name(struct kobject *kobj, const char *fmt, ...);
```

其各个参数及返回值的含义解释如下。

- ◆ **kobj**: 指向要设置名称的内核对象。
- ◆ **fmt**: 格式字符串, 它与后面的可变参数结合, 以类似于 **printf** 函数的方式构造内核对象的名称。
- ◆ **返回值**: 0 表示成功, 负数为错误码。

注意由于涉及到内存的动态分配和释放, 不能直接给内核对象的 **name** 成员赋值。由于内核对象的名称在注册后将作为目录名, 因此不要包含斜杠。

以上函数不能用来设置已注册的内核对象的名称, 这种情况下必须用内核对象的改名函数, 原型如下:

```
int kobject_rename(struct kobject *kobj, const char *name);
```

其各个参数及返回值的含义解释如下。

- ◆ **kobj**: 指向要设置名称的内核对象。
- ◆ **name**: 内核对象的新名称。
- ◆ **返回值**: 0 表示成功, 负数为错误码。

使用下面这个函数可以得到内核对象的名称:

```
const char *kobject_name(const struct kobject *kobj);
```

其中参数 **kobj** 指向一个内核对象, 返回值是它的名称。

15.1.2.3 内核对象的初始化与注册

内核对象可以用以下函数加以初始化:

```
void kobject_init(struct kobject *kobj, struct kobj_type *ktype);
```

其各个参数的含义解释如下。

- ◆ **kobj**: 指向要初始化的 **kobj**。
- ◆ **ktype**: 一个指针, 用于设置内核对象的类型。

初始化后的内核对象可以向内核注册，注册的结果是在 `sysfs` 伪文件系统中出现一个以对象名称命名的目录。注册的基本接口函数原型如下：

```
int kobject_add(struct kobject *kobj, struct kobject *parent,
               const char *fmt, ...);
```

其各个参数与返回值的含义解释如下。

- ◆ `kobj`：指向要注册的内核对象。
- ◆ `parent`：指定内核对象的父对象，可以为 `NULL`。
- ◆ `fmt`：格式字符串，它与后面的可变参数结合以构造内核对象的名称。
- ◆ 返回值：`0` 表示成功，负数为错误码。

注册时可以指定内核对象的父对象，对应的目录会建立在父对象的目录中；如果指定为 `NULL`，则它的父对象会被设为其 `kset` 成员指向的内核集合中嵌套的内核对象；如果其 `kset` 成员也设为 `NULL`，则对应的目录出现在 `sysfs` 伪文件系统的根目录下。另外注册时还会增加对父对象的引用计数。

`sysfs` 伪文件系统通常挂载在 `/sys` 目录下。

另外一个函数将初始化和注册操作结合起来，原型如下：

```
int kobject_init_and_add(struct kobject *kobj, struct kobj_type *ktype,
                        struct kobject *parent, const char *fmt, ...);
```

它的参数也是 `kobject_init` 和 `kobject_add` 两个函数参数的综合。

内核对象也可以动态创建，使用下面的函数：

```
struct kobject *kobject_create(void);
```

其返回值指向新创建的内核对象，如果创建失败，则返回 `NULL`。但内核源码中并没有为这个函数导出符号，所以不能直接在模块编程时使用。

创建操作也可以和注册操作结合起来，用下面这个函数：

```
struct kobject *kobject_create_and_add(const char *name,
                                       struct kobject *parent);
```

其各个参数及返回值的含义解释如下。

- ◆ `name`：内核对象的名称。
- ◆ `parent`：指定内核对象的父对象，可以为 `NULL`。
- ◆ 返回值：指向新创建的内核对象，`NULL` 表示创建或注册失败。

这个函数的符号已经导出，可以直接在模块编程时使用。但是用这种方法创建的内核对象由于无法在注册前设置 `kset` 指针，因此无法加入某个内核集合。

与注册相反的操作是注销，用下面的函数：

```
void kobject_del(struct kobject *kobj);
```

其中参数 `kobj` 指向要注销的内核对象。注销时 `sysfs` 伪文件系统中的相应目录将被删除，对父对象的引用计数也要减少。这个函数一般不直接调用，因为内核对象的引用计数降到 0 之后就会自动执行注销操作，因此只需要对它的引用计数进行操作，而把何时该注销的问题交给内核判断。

15.1.2.4 内核对象的引用计数

由于内核对象内嵌了一个引用计数成员，所以很容易实现引用计数的相关操作。
内核对象的获取操作定义如下：

```
struct kobject *kobject_get(struct kobject *kobj)
{
    if (kobj) kref_get(&kobj->kref);
    return kobj;
}
```

内核对象的释放操作定义如下：

```
void kobject_put(struct kobject *kobj)
{
    if (kobj) kref_put(&kobj->kref, kobject_release);
}
```

其中的 `kobject_release` 是内核为对象内嵌的 `kref` 成员定义的释放函数。

15.1.3 内核对象的类型

这里的类型不是指内核对象的数据类型，而是它内嵌的 `(struct kobj_type *)` 型指针，这个指针指向的结构体所描述的是这个内核对象的类型，它的定义如下：

```
struct kobj_type {
    void (*release)(struct kobject *kobj); /* 内核对象的释放函数 */
    struct sysfs_ops *sysfs_ops; /* sysfs 文件操作 */
    struct attribute **default_attrs; /* 内核对象的默认属性 */
};
```

其中的成员 `release` 是一个函数指针，它会在函数 `kobject_release` 里被回调，可以把它认为是内核对象自己的释放函数。`sysfs_ops` 和 `default_attrs` 成员组合起来用于实现内核对象的属性。

15.1.3.1 内核对象的属性

`default_attrs` 指向一个指针数组，这个数组中的每个指针指向一个用于描述对象属性的数据，这个数据的类型定义如下：

```
struct attribute {
    const char *name; /* 属性的名称 */
    mode_t mode; /* 属性的访问权限 */
};
```

其中的 `name` 成员是一个字符串，表示属性的名称，而 `mode` 成员则表示访问权限，它是 `mode_t` 型数据，实际定义如下：

```
typedef unsigned short    __kernel_mode_t;
typedef __kernel_mode_t  mode_t;
```

内核对象注册以后，它的默认属性由成员 `ktype` 指向的类型中的 `default_attrs` 成员决定，每个属性将呈现为内核对象目录下的一个文件，文件名就是属性名，而文件的访问权限就是属性的访问权限。注意 `default_attrs` 指向的数组的最后一个元素必须是 `NULL`，内核据此来判断所有的默认属性已描述完毕。

属性既然表现为文件的形式，那么一定可以被读写。当应用程序读写属性所对应的文件时，内核将回调由成员 `sysfs_ops` 指向的操作，它的定义如下：

```
struct sysfs_ops {
    ssize_t (*show)(struct kobject *kobj, struct attribute *attr,
                    char *buf);
    ssize_t (*store)(struct kobject *kobj, struct attribute *attr,
                    const char *buf, size_t size);
};
```

这个结构体只有两个成员，都是函数指针，其中成员 `show` 指向的函数将会在应用程序读属性文件时被调用，成员 `store` 指向的函数将会在应用程序写属性文件时被调用，两者被调用时都会将对应的内核对象和属性的指针传入。

与文件的读操作相比，这里的 `show` 操作只传入了一个缓冲区地址，没有传入缓冲区长度。实际上，`buf` 参数指向的缓冲区是由内核自动分配的，长度是一页内存，一般是 **4KB**。它不是用户态内存指针，所以可以直接访问。向这个缓冲区写入数据后，返回数据的长度，应用程序就可以得到这些数据。`store` 操作与此类似，这里传入的 `buf` 参数也是内核态内存指针，`size` 参数则是数据的长度，返回值是实际写入的数据长度。`show` 和 `store` 操作必须根据传入的 `attr` 参数来判断应用程序读的是哪一个属性，然后做相应的操作。

15.1.3.2 动态创建属性

内核对象除了有默认属性之外，还可以在注册后动态创建属性，用下面这个函数：

```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr);
```

其各个参数及返回值的含义解释如下。

- ◆ `kobj`：指向要创建属性的内核对象（目录）。
- ◆ `attr`：指向要创建的新属性（文件）。
- ◆ 返回值：0 表示成功，负数为错误码。

动态创建的属性可以被删除，用下面这个函数：

```
void sysfs_remove_file(struct kobject *kobj, const struct attribute *attr);
```

其各个参数的含义解释如下。

- ◆ **kobj**: 指向属性所属的内核对象（所在的目录）。
- ◆ **attr**: 指向要删除的属性（文件）。

动态创建属性相当于在内核对象的目录中动态新建一个文件，当应用程序读写这个文件时，内核仍然调用原来注册内核对象时提供的 **show** 和 **store** 操作，因此这些操作函数在实现时就应该考虑到将来可能会动态创建的属性。

但是在很多情况下，内核对象的 **show** 和 **store** 操作无法未卜先知将来可能动态创建的属性，这时可以考虑对 **sysfs_ops** 结构体的两个操作函数和 **attribute** 结构体进行包装，使之通用。思路是这样的，属性与读写属性时用的操作应该绑定在一起，于是定义以下的数据类型：

```
struct kobj_attribute {
    struct attribute attr;
    ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr,
        char *buf);
    ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr,
        const char *buf, size_t count);
};
```

这样，每定义一个属性，就必须伴随着定义一个 **show** 操作和 **store** 操作，注意它们的参数与 **sysfs_ops** 结构体中的 **show** 和 **store** 操作并不相同。

然后定义一个通用的 **show** 操作如下：

```
static ssize_t kobj_attr_show(struct kobject *kobj, struct attribute *attr,
    char *buf)
{
    struct kobj_attribute *kattrib;
    ssize_t ret = -EIO;
    kattrib = container_of(attr, struct kobj_attribute, attr);
    if (kattrib->show) ret = kattrib->show(kobj, kattrib, buf);
    return ret;
}
```

将这个函数的入口地址赋值给 **struct sysfs_ops** 的 **show** 指针，因此当应用程序读内核对象属性时，内核会调用这个函数。在这个函数里，首先利用 **container_of** 宏从指向 **struct attribute** 的指针得到指向外围的 **struct kobj_attribute** 的指针，然后就可以将调用传递给伴随着属性提供的 **show** 方法。

通用的 **store** 操作的实现与此类似：

```
static ssize_t kobj_attr_store(struct kobject *kobj, struct attribute *attr,
    const char *buf, size_t count)
{
    struct kobj_attribute *kattrib;
    ssize_t ret = -EIO;
    kattrib = container_of(attr, struct kobj_attribute, attr);
    if (kattrib->store) ret = kattrib->store(kobj, kattrib, buf, count);
    return ret;
}
```


将通用的 `show` 和 `store` 操作放在一个 `sysfs_ops` 结构体内以方便使用：

```
struct sysfs_ops kobj_sysfs_ops = {
    .show = kobj_attr_show,
    .store = kobj_attr_store,
};
```

实际上这些代码都已经存在于内核源码中，但并未导出符号，所以不能直接在模块编程时引用它们。

内核还定义了一个宏专门用于初始化与 `struct kobj_attribute` 类似的数据类型：

```
__ATTR(name, mode, show, store);
```

其各个参数的含义解释如下。

- ◆ `name`：属性的名称，会自动展开成字符串，不要再加双引号。
- ◆ `mode`：属性的访问权限。
- ◆ `show`：读属性时的操作。
- ◆ `store`：写属性时的操作。

于是就可以用如下的语句定义内核对象的属性：

```
struct kobj_attribute attr_xxx = __ATTR(xxx, S_IRUGO, xxx_show, xxx_store);
```



由于宏展开时不检查参数类型，所以在内核源码中 `__ATTR` 宏被普遍应用于各种相似数据类型的变量初始化，只要它们有与 `struct kobj_attribute` 相同名称的成员即可，成员的类型不必相同。

图 15.1 示意了内核对象属性的实现过程：首先定义一个内核对象变量 `kobj`，它的 `ktype` 指针指向一个已定义的用于描述对象类型的变量 `ktype`；变量 `ktype` 的 `sysfs_ops` 指针指向 `kobj_sysfs_ops` 变量，`default_attrs` 指针则指向一个数组 `attrs[]`；数组 `attrs[]` 的各个元素都是指针，除最后一个元素必须是 `NULL` 外，其他元素分别指向已定义的用于描述对象属性的两个变量；这两个变量中的 `show` 和 `store` 指针指向的函数将会在 `kobj_attr_show` 函数和 `kobj_attr_store` 函数中被回调，最终实现对属性文件的读写操作。

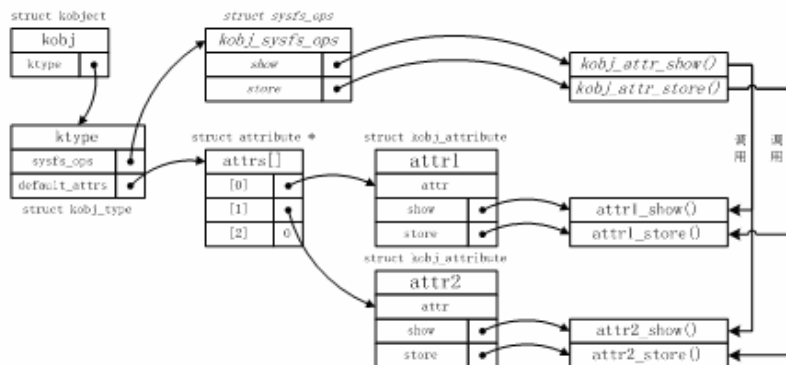


图 15.1 内核对象属性的实现过程

可以注意到，在动态创建内核对象时，并没有要求提供对象类型。其实内核为它提供了一个内在的类型，定义如下：

```
static struct kobj_type dynamic_kobj_ktype = {
    .release = dynamic_kobj_release, /* 动态创建对象的释放函数 */
    .sysfs_ops = &kobj_sysfs_ops, /* 内核对象的通用 sysfs 操作 */
};
```

可见它用了内核对象的通用 `sysfs` 操作，这样就可以用上述包装过的接口为它动态创建属性了。它的释放函数定义如下：

```
static void dynamic_kobj_release(struct kobject *kobj)
{
    kfree(kobj);
}
```

也就是说，当引用计数降为 0 后，动态创建的内核对象会自动释放内存空间。

15.1.4 内核集合

内核集合是基于内核对象而设计的，它可以收纳内核对象，并且还可以管理内核对象所发送的用户态事件。

15.1.4.1 内核集合数据类型

内核集合由如下的结构体表示：

```
struct kset {
    struct list_head list; /* 链表头 */
    spinlock_t list_lock; /* 自旋锁 */
    struct kobject kobj; /* 内嵌的内核对象 */
    struct kset_uevent_ops *uevent_ops; /* 指向用户态事件操作 */
};
```

其中各个成员的含义解释如下。

- ◆ `list`：这是一个链表头，它用来链住所有属于这个集合的内核对象。
- ◆ `list_lock`：自旋锁，用于访问链表时的同步。
- ◆ `kobj`：内核对象，集合的注册、注销等操作以及属性等特征由它作为代表实现。
- ◆ `uevent_ops`：指向一个结构体，其中包含各种用户态事件的操作函数指针。

15.1.4.2 内核集合的注册和注销

用下面的函数可以向内核注册一个内核集合：

```
int kset_register(struct kset *kset);
```

其参数和返回值的含义解释如下。

- ◆ `kset`：指向被注册的内核集合。

◆ 返回值：0 表示成功，负数为错误码。

这个函数在注册内核集合前会先将其初始化。实际上，注册操作是将内核集合内嵌的 `kobj` 成员注册，因此也会在 `sysfs` 伪文件系统里出现一个目录，目录名是 `kobj` 的名称，目录内的文件是 `kobj` 的属性。注册了的内核集合有时又称为子系统。

注册以后的内核集合可以用下面的函数注销：

```
void kset_unregister(struct kset *kset);
```

其中参数 `kset` 指向要注销的内核集合。这里只对内核集合的 `kobj` 成员做了一个释放操作。内核集合也有一个动态创建并注册的操作，其接口函数原型如下：

```
struct kset *kset_create_and_add(const char *name,
                                struct kset_uevent_ops *u, struct kobject *parent_kobj);
```

其各个参数和返回值的含义解释如下。

- ◆ `name`：内核集合的名称。
- ◆ `u`：指向用户态事件的操作函数。
- ◆ `parent_kobj`：内核集合作为对象出现时的父对象。
- ◆ 返回值：指向新创建的内核集合，`NULL` 表示创建或注册失败。

动态创建的内核集合中内嵌的内核对象也用 `kobj_sysfs_ops` 作为它的属性操作，因此也可以动态创建属性。

15.1.4.3 内核集合的引用计数

因为经常要由内核集合内嵌的内核对象的指针得到指向它自己的指针，所以内核定义了以下函数：

```
static inline struct kset *to_kset(struct kobject *kobj)
{
    return kobj ? container_of(kobj, struct kset, kobj) : NULL;
}
```

在此基础上实现了内核集合的获取与释放操作：

```
static inline struct kset *kset_get(struct kset *k)
{
    return k ? to_kset(kobject_get(&k->kobj)) : NULL;
}

static inline void kset_put(struct kset *k)
{
    kobject_put(&k->kobj);
}
```

可以看出，这里几乎是原封不动地“继承”了内核对象的相关操作。

15.1.4.4 用户态事件

注册了的内核对象可以发送用户态事件，其接口函数原型如下：

```
int kobject_uevent_env(struct kobject *kobj,
                      enum kobject_action action, char *envp[]);
```

其各个参数和返回值的含义解释如下。

- ◆ **kobj**: 指向发送事件的内核对象。
- ◆ **action**: 事件的类型。
- ◆ **envp**: 指向一个以 **NULL** 结尾的字符串数组，其中每个字符串是一条环境变量的设置信息，这些环境变量添加在默认的环境变量之后。
- ◆ **返回值**: 0 表示成功，负数为错误码。

如果不需要添加额外的环境变量，可以用一个由上述函数包装而成的更简洁的函数：

```
int kobject_uevent(struct kobject *kobj, enum kobject_action action)
{
    return kobject_uevent_env(kobj, action, NULL);
}
```

表示事件类型的参数是枚举型，目前包含以下的可选值：

```
enum kobject_action {
    KOBJ_ADD, /* 注册 */
    KOBJ_REMOVE, /* 注销 */
    KOBJ_CHANGE, /* 修改 */
    KOBJ_MOVE, /* 移动 */
    KOBJ_ONLINE, /* 在线 */
    KOBJ_OFFLINE, /* 离线 */
    KOBJ_MAX /* 不是一种动作，仅用来表示可选值的个数 */
};
```

当一个内核集合被注册时，它内嵌的内核对象将自动发送一个 **KOBJ_ADD** 事件，相应地在注销时自动发送一个 **KOBJ_REMOVE** 事件，当内核对象被改名或者移动时，将自动发送一个 **KOBJ_MOVE** 事件，这些事件的发送是默认的。

发送用户态事件时，内核将默认设置以下环境变量。

- ◆ **ACTION**: 表示事件的类型。
- ◆ **DEVPATH**: 表示内核对象在 **sysfs** 文件系统中的绝对路径。
- ◆ **SUBSYSTEM**: 子系统名，默认是内核对象所属集合的名称。
- ◆ **SEQNUM**: 流水号，每发送一个事件，这个数值就会加 1。

发送的用户态事件将以两种方式告知应用程序，下面分别对它们进行讲述。

一种是用户态帮手（**user mode helper**）的方式。在这种方式下，内核将直接执行一个指定的应用程序，并给它传递相应环境变量。内核执行这个程序时，还会为其多设置两个环境变量如下：

```
HOME=/
PATH=/sbin:/bin:/usr/sbin:/usr/bin
```

这个应用程序可以在编译内核时由以下配置选项指定：

```
CONFIG_UEVENT_HELPER_PATH="/sbin/hotplug"
```

这里给的是默认值，它还可以在系统运行时修改，方法是修改 `kernel` 子系统的属性 `uevent_helper`，即文件 `/sys/kernel/uevent_helper`，或者修改 `proc` 伪文件系统中的文件 `/proc/sys/kernel/hotplug`。

这种方法的缺点是内核的功能依赖于一个应用程序，对其稳定性及源码的可维护性都不利，因此现在各种 Linux 发行版都倾向于用 `NETLINK` 套接字的办法。`NETLINK` 是一种内核态与用户态通信的机制。在应用程序内，可以直接用套接字接口打开 `NETLINK` 连接，如：

```
netlink_socket = socket(PF_NETLINK, SOCK_DGRAM, NETLINK_KOBJECT_UEVENT);
```

这行代码打开了一个用于接收内核对象的用户态事件的套接字，随后的操作都与普通的套接字类似。在内核里，事件的类型、所在的子系统以及各个环境变量等信息会被组合成一个字符串，伪装成网络接口收到的数据传给应用程序。这里，所谓的环境变量已经失去了它本来的意义。

在 Linux 2.6.30 内核中，如果内核的配置选项支持网络接口，则同时用两种方式将事件通知到用户态。

所发送的用户态事件并不一定能一帆风顺地到达应用程序那里。首先发送事件的内核对象必须属于某个集合，其次它的 `uevent_suppress` 标志必须是 0，最后它发送的事件还要经过所属内核集合的过滤操作，这些操作由内核集合的 `uevent_ops` 成员指针确定，定义如下：

```
struct kset_uevent_ops {
    int (*filter)(struct kset *kset, struct kobject *kobj);
    const char *(*name)(struct kset *kset, struct kobject *kobj);
    int (*uevent)(struct kset *kset, struct kobject *kobj,
        struct kobj_uevent_env *env);
};
```

这是三个函数指针，其中 `filter` 指向的函数用做过滤器，当这个函数返回 0 时，事件就被丢弃；`name` 则用于子系统名称的过滤，`SUBSYSTEM` 环境变量的值会被修改为它的返回值；`uevent` 指向的函数则是事件的钩子函数，内核集合可以在其中做一些额外的操作，如果这个函数返回非 0 值，则事件也会被丢弃。

15.1.4.5 内核集合与内核对象

内核对象在注册前如果设置了 `kset` 指针，则注册时它就会被加入到 `kset` 指针所指向的内核集合中，它所发送的用户态事件将接受这个内核集合的过滤操作。

内核集合本身有一个内嵌的内核对象，因此也可以加入到其他集合中。

内核集合与它收纳的内核对象的链接关系可由图 15.2 示意，其中集合 `kset` 收纳了三个内核对象 `kobj1`，`kobj2` 和 `kobj3`，它们都放在 `kset` 的 `list` 链表中，三个对象的 `kset` 指针都指向集合 `kset`，对象 `kobj2` 的父对象是 `kobj1`，而对象 `kobj1` 和 `kobj2` 的父对象都是集合 `kset` 内嵌的成员对象 `kobj`。

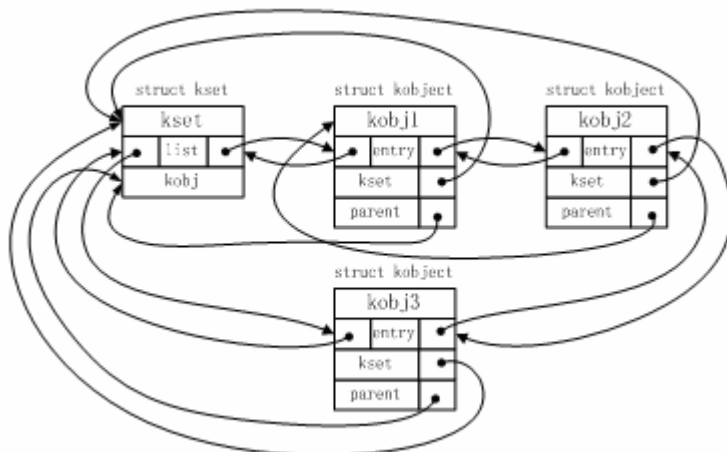


图 15.2 内核集合与内核对象

15.1.5 内核集合与对象例程

为了加深对设备模型的基本机制的理解，我们将编写一个例程。这个例程在加载时首先创建并注册了一个顶级子系统 **test**，然后在这个子系统内注册了若干个内核对象，每个对象有两个默认属性：**name** 属性为只读属性，返回对象的名称；**string** 属性为可写属性，可以把写入的数据保存成一个字符串，读取时将返回这个字符串。注意所有对象的 **string** 属性的值都保存在同一个字符串中。

15.1.5.1 源码

例程源码如下：

```
/* 文件名: sysfs_test.c */
/* 说明: 内核集合与对象例程 */

#include <linux/module.h>

MODULE_LICENSE("GPL");

#define SZ_STRING 30 /* 字符串的最大长度 */
#define NR_KOBS 3 /* 内核对象的个数 */

/* 仿照内核的 kobj_attr_show 函数 */
static ssize_t test_attr_show(struct kobject *kobj, struct attribute *attr,
                             char *buf)
{
    struct kobj_attribute *kattrib;
    ssize_t ret = -EIO;
    kattrib = container_of(attr, struct kobj_attribute, attr);
    if (kattrib->show)
        ret = kattrib->show(kobj, kattrib, buf);
    return ret;
}
```

```

/* 仿照内核的 kobj_attr_store 函数 */
static ssize_t test_attr_store(struct kobject *kobj, struct attribute *attr,
                               const char *buf, size_t count)
{
    struct kobj_attribute *kattr;
    ssize_t ret = -EIO;
    kattr = container_of(attr, struct kobj_attribute, attr);
    if (kattr->store)
        ret = kattr->store(kobj, kattr, buf, count);
    return ret;
}

/* 仿照内核的 kobj_sysfs_ops 变量 */
static struct sysfs_ops test_sysfs_ops = {
    .show    = test_attr_show,
    .store    = test_attr_store,
};

/* 内核对象的释放函数 */
static void test_release(struct kobject *kobj)
{
    kfree(kobj);
}

/* 属性 name 的 show 函数 */
static ssize_t test_name_show(struct kobject *kobj,
                              struct kobj_attribute *attr, char *buf)
{
    return sprintf(buf, "%s\n", kobject_name(kobj));
}

/* 属性 name 的 store 函数, 因为是只读属性, 因此直接返回错误 */
static ssize_t test_name_store(struct kobject *kobj,
                               struct kobj_attribute *attr, const char *buf, size_t len)
{
    return -EACCES; /* 错误号表示无权限 */
}

/* 属性 name 的定义 */
static struct kobj_attribute test_attr_name =
    __ATTR(name, S_IRUGO, test_name_show, test_name_store);

/* 用于保存属性 string 对应的字符串 */
static char test_string[SZ_STRING] = "test\n";

/* 属性 string 的 show 函数 */
static ssize_t test_string_show(struct kobject *kobj,
                                struct kobj_attribute *attr, char *buf)
{
    return sprintf(buf, test_string);
}

```

```

/* 属性 string 的 store 函数 */
static ssize_t test_string_store(struct kobject *kobj,
                                struct kobj_attribute *attr, const char *buf, size_t len)
{
    if (len >= SZ_STRING) len = SZ_STRING-1; /* 限制写的长度 */
    memcpy(test_string, buf, len); /* 复制数据 */
    test_string[len] = '\0'; /* 字符串结尾 */
    return len;
}

/* 属性 string 的定义 */
static struct kobj_attribute test_attr_string =
    __ATTR(string, S_IRUGO|S_IWUGO, test_string_show, test_string_store);

/* 对象的默认属性 */
static struct attribute *test_default_attr[] = {
    &test_attr_name.attr,
    &test_attr_string.attr,
    NULL,
};

/* 对象的类型 */
static struct kobj_type test_kobj_type = {
    .release = test_release, /* 释放函数 */
    .sysfs_ops = &test_sysfs_ops, /* 属性操作 */
    .default_attrs = test_default_attr /* 默认属性 */
};

/* 用于保存动态创建的内核集合的指针 */
static struct kset *kset;

/* 模块的初始化函数 */
static __init int sysfs_test_init(void)
{
    int i; /* 循环变量 */
    int err; /* 用于暂存错误码 */
    struct kobject *kobj, *next; /* 循环变量 */
    pr_debug("sysfs_test_init: be called.\n");
    /* 创建并注册内核集合, 由于没有父对象, 将成为顶级子系统 */
    if ((kset = kset_create_and_add("test", NULL, NULL)) == NULL) {
        pr_debug("sysfs_test_init: kset_create_and_add ERR!\n");
        err = -ENOMEM;
        goto kset_fail;
    }
    /* 为 kset 集合动态创建一个属性, 使用 name 属性的定义 */
    if ((err = sysfs_create_file(&kset->kobj, &test_attr_name.attr)) {
        pr_debug("sysfs_test_init: sysfs_create_file ERR!\n");
        goto sysfs_fail;
    }
    /* 注册 NR_KOBS 个内核对象, 都属于 kset 集合 */

```



```

    for (i = 0; i < NR_KOBS; i++) {
        /* 分配内存 */
        kobj = kzalloc(sizeof(struct kobject), GFP_KERNEL);
        if (!kobj) {
            pr_debug("sysfs_test_init: kzalloc ERR!\n");
            err = -ENOMEM;
            goto kobject_fail;
        }
        /* 设置所属集合 */
        kobj->kset = kset;
        /* 初始化并注册 */
        err = kobject_init_and_add(kobj, &test_kobj_type, NULL, "obj%d", i);
        if (err) {
            pr_debug("sysfs_test_init: kobject_register ERR = %d\n", err);
            kfree(kobj);
            goto kobject_fail;
        }
        /* 发送事件 */
        kobject_uevent(kobj, KOBJ_ADD);
    }
    return 0;
kobject_fail:
    list_for_each_entry_safe(kobj, next, &kset->list, entry)
        kobject_put(kobj);
sysfs_fail:
    kset_unregister(kset);
kset_fail:
    return err;
}
module_init(sysfs_test_init);

/* 模块的退出函数 */
static __exit void sysfs_test_exit(void)
{
    struct kobject *kobj, *next;
    pr_debug("sysfs_test_exit: be called.\n");
    /* 首先清理集合中的所有对象 */
    list_for_each_entry_safe(kobj, next, &kset->list, entry)
        kobject_put(kobj);
    /* 然后注销集合 */
    kset_unregister(kset);
}
module_exit(sysfs_test_exit);

```

15.1.5.2 说明

为了捕捉到用户态事件，可以写一个脚本，并把它设置为用户态帮手，用类似如下的命令：

```
echo ${PWD}/uhelper.sh > /proc/sys/kernel/hotplug
```

其中 `uhelper.sh` 是脚本的名称。注意当内核执行它时并不在当前用户的会话中，所以无法直

接向屏幕输出信息，但我们可以利用 `write` 命令向指定用户发送信息，于是设计 `uhelper.sh` 的内容如下：

```
#!/bin/sh
write cjhy <<EOF # 输出到用户 cjhy 的控制台上
ACTION=${ACTION} DEVPATH=${DEVPATH} SUBSYSTEM=${SUBSYSTEM} SEQNUM=${SEQNUM}
EOF
```

如果是在目标机上，有可能不支持 `write` 命令，但因为进行嵌入式系统调试时，使用的就是默认终端，因此可以将信息重定向到 `/dev/console`，如：

```
#!/bin/sh
echo ACTION=${ACTION} DEVPATH=${DEVPATH} SUBSYSTEM=${SUBSYSTEM} SEQNUM=${SEQNUM}
> /dev/console
```

这样，当加载模块时就会在终端上出现类似以下的信息：

```
ACTION=add DEVPATH=/module/sysfs_test SUBSYSTEM=module SEQNUM=924
ACTION=add DEVPATH=/test/obj0 SUBSYSTEM=test SEQNUM=925
ACTION=add DEVPATH=/test/obj1 SUBSYSTEM=test SEQNUM=926
ACTION=add DEVPATH=/test/obj2 SUBSYSTEM=test SEQNUM=927
```

从中可以看到，模块加载时本身就会在 `module` 子系统中发出 `add` 事件。
本例程在 `Linux 2.6.26` 和 `Linux 2.6.30` 内核上编译并测试通过。

15.2 设备管理

从内核对象、内核集合这两个基本数据类型出发，内核又定义了许多有实际含义的数据类型，如设备、驱动、总线等，这些数据类型的定义及相关接口的声明在以下头文件中：

```
#include <linux/device.h>
```

内核在启动时就会注册多个子系统，用于设备、驱动、总线等的管理，几个主要的顶级子系统如下。

- ◆ `devices`：顶级子系统，用于收纳各种设备。
- ◆ `bus`：顶级子系统，用于收纳各种总线。
- ◆ `class`：顶级子系统，用于收纳各种类别。

15.2.1 设备

在 `devices` 子系统中收纳的是各种设备。这里的设备指的是设备模型中的一种数据对象，它是具体设备的一种抽象。

15.2.1.1 设备数据类型

表示设备的数据类型定义如下：

```

struct device {
    struct device *parent; /* 指向父设备 */
    struct device_private *p; /* 指向内核内部使用的设备数据 */
    struct kobject kobj; /* 内嵌的内核对象 */
    struct device_type *type; /* 指向设备的类型，描述了设备的属性和操作 */
    struct bus_type *bus; /* 指向设备所属的总线 */
    struct device_driver *driver; /* 指向与设备绑定的驱动 */
    void *driver_data; /* 指向驱动的私有数据 */
    void *platform_data; /* 指向传给驱动的平台相关的私有数据 */
    dev_t devt; /* 设备号 */
    struct class *class; /* 指向设备所属的类别 */
    void (*release)(struct device *dev); /* 设备的释放函数 */
    /* 这里只写出了部分成员 */
};

```

其中的 `struct device_type` 用来表示设备的类型，它的定义如下：

```

struct device_type {
    const char *name; /* 设备类型的名称 */
    int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
    void (*release)(struct device *dev);
};

```

可见，设备有以下基本特征：设备号、所属的总线、所绑定的驱动、所属的类别等。当然，并不是每个设备都同时具有这些特征。设备也有父子关系，用于描述设备之间的连接关系和从属关系。当设备发送用户态事件时，`devices` 子系统的用户态事件钩子函数给它增加了以下环境变量。

- ◆ **MAJOR**：主设备号，如果设备号为 0，则不设置这个环境变量。
- ◆ **MINOR**：次设备号，如果设备号为 0，则不设置这个环境变量。
- ◆ **DEVTYPE**：设备类型的名称。
- ◆ **DRIVER**：所绑定的驱动的名称。

这个函数同时还回调设备类型中的 `uevent` 函数，因此可以在这里增加设备特有的环境变量。

当设备的引用计数降为 0 时，会调用设备的 `release` 操作；如果设备没有 `release` 操作，则调用设备类型的 `release` 操作。

15.2.1.2 设备的注册与注销

下面是设备的注册和注销操作的函数原型：

```

int device_register(struct device *dev); /* 注册设备 */
void device_unregister(struct device *dev); /* 注销设备 */

```

注册设备将使它出现在 `devices` 子系统中，设备对应的目录在它的父设备的目录下。注册设备的同时还进行以下几个主要操作。

- ◆ 如果有设备号，则建立一个 `dev` 只读属性。
- ◆ 如果有所属类别，则在这个类别对应的目录中建立一个符号链接指向这个设备对应的目录。

- ◆ 如果有所属总线,则在这个总线的 **devices** 集合对应的目录中建立一个符号链接指向这个设备对应的目录。
- ◆ 如果所属总线的自动探测功能是打开的,则与这个总线上注册的驱动逐个进行关联操作。

15.2.1.3 设备的引用计数

设备也支持引用计数的相关操作,函数原型如下:

```
struct device *get_device(struct device *dev); /* 增加引用计数 */
void put_device(struct device *dev); /* 减少引用计数 */
```

15.2.1.4 私有数据

驱动的私有数据可以用下面的函数与设备绑定在一起:

```
static inline void *dev_get_drvdata(const struct device *dev)
{
    return dev->driver_data;
}

static inline void dev_set_drvdata(struct device *dev, void *data)
{
    dev->driver_data = data;
}
```

15.2.1.5 设备的创建与销毁

为了方便使用,内核还提供了动态创建设备并注册的接口函数,原型如下:

```
struct device *device_create(struct class *cls, struct device *parent,
                             dev_t devt, void *drvdata, const char *fmt, ...);
```

其各个参数及返回值的含义解释如下。

- ◆ **cls**: 指向设备所属的类别。
- ◆ **parent**: 指向父设备。
- ◆ **devt**: 设备号。
- ◆ **drvdata**: 指向给驱动的私有数据。
- ◆ **fmt**: 格式字符串,与后面的可变参数结合,用于设置设备的名称。
- ◆ **返回值**: 指向新创建的设备,创建或注册失败时为包装成指针的错误码。

这个函数只能用于创建属于某个类别的设备,并且必须提供一个设备号,所以一般用于为字符设备创建对应的设备模型中的设备,以纳入设备模型的管理体系。



当 `device_create` 函数创建设备失败时,返回值也不是 `NULL`。

与创建设备相反的操作是销毁设备,其接口函数原型如下:

```
void device_destroy(struct class *cls, dev_t devt);
```

其中参数 `cls` 指向设备所属的类别，`devt` 是设备号。注意销毁时并不需要提供设备自己的地址，实际上内核是从类别中通过设备号查找到设备的，因此在创建设备时就必须有类别和设备号。

15.2.2 错误码与指针

通常情况下，返回指针的函数用 `NULL` 表示操作失败，但如果希望在失败时能返回相应的错误码，则需要将错误码包装成指针值，而且这个指针值绝对不能是有效的地址。

内核有足够的无效地址空间来包装错误码。错误码与指针值相互转换的接口放在以下头文件中：

```
#include <linux/err.h>
```

下面这个函数可以将错误码转换成指针值：

```
void *ERR_PTR(long error);
```

注意这里的错误码 `error` 必须是一个负数，且大于 `-4096`。

下面这个函数可以将指针值转换成错误码：

```
long PTR_ERR(const void *ptr);
```

下面这个函数用来判断指针值是否代表错误码：

```
long IS_ERR(const void *ptr);
```

实际应用时，返回指针的函数可用 `ERR_PTR` 函数将错误码包装成指针值后返回，调用者必须先用 `IS_ERR` 判断得到的指针值是否代表错误码，如果是错误码则可以用 `PTR_ERR` 函数将指针值转换成错误码，注意转换得到的错误码也是一个负数。

15.2.3 驱动

这里的驱动指的是设备模型中的一种数据对象，它的主要作用是向总线提供设备注册和注销时的操作。

15.2.3.1 驱动数据类型

表示驱动的数据类型定义如下：

```
struct device_driver {
    const char *name; /* 驱动的名称 */
    struct bus_type *bus; /* 驱动所属的总线 */
    struct module *owner; /* 驱动所属的模块 */
    int (*probe)(struct device *dev); /* 探测操作 */
    int (*remove)(struct device *dev); /* 移除操作 */
    void (*shutdown)(struct device *dev); /* 关机操作 */
    int (*suspend)(struct device *dev, pm_message_t state); /* 挂起操作 */
    int (*resume)(struct device *dev); /* 恢复操作 */
    struct driver_private *p; /* 指向内核内部使用的驱动相关数据 */
    /* 这里只写出了部分成员 */
};
```

```
};
```

这个类型的变量总是静态定义的，其中 `struct driver_private` 类型的定义如下：

```
struct driver_private {
    struct kobject kobj; /* 内核对象 */
    struct device_driver *driver; /* 指向所联系的驱动 */
    /* 这里只写出了部分成员 */
};
```

可见驱动数据类型最终也跟一个内核对象联系在一起，只不过这些数据域是仅在设备模型代码内部使用的，内核不希望被其他模块访问，所以特意从所联系的数据类型中剥离了出来。在驱动注册时，内核会自动创建一个 `struct driver_private` 型数据并把它与驱动联系在一起。

驱动有两个必需的属性：名称和所属总线。其他的重要成员都是一些函数指针，代表不同的操作，其中最重要的两个操作是 `probe` 操作和 `remove` 操作。`probe` 操作在驱动与设备关联的过程中被调用，`remove` 操作则在注销设备或者注销驱动时被调用。

当系统关机或重启时，驱动的 `shutdown` 操作将被调用。`suspend` 和 `resume` 操作则与系统的电源管理有关。当设备进入挂起状态时调用 `suspend` 操作，当设备从挂起状态恢复时调用 `resume` 操作。

15.2.3.2 驱动的注册与注销

下面是驱动的注册与注销函数的原型：

```
int driver_register(struct device_driver *drv); /* 注册驱动 */
void driver_unregister(struct device_driver *drv); /* 注销驱动 */
```

注册一个驱动将使它出现在所属总线的 `drivers` 集合内，如果所属总线的自动探测功能是打开的，则还会与总线上注册的设备逐个进行关联操作。

15.2.3.3 驱动的引用计数

驱动也支持引用计数的相关操作，相关函数原型如下：

```
struct device_driver *get_driver(struct device_driver *drv); /* 增加引用计数 */
void put_driver(struct device_driver *drv); /* 减少引用计数 */
```

15.2.4 总线

在 `bus` 子系统中收纳的都是各种总线。总线能够接受设备和驱动的注册，并且进行设备和驱动的关联操作。通过总线，设备和驱动就可以分离，并且可以各自独立地进行动态的注册和注销操作。

15.2.4.1 总线数据结构

表示总线的数据类型定义如下：

```
struct bus_type {
    const char *name; /* 总线的名称 */
```

```

int (*match)(struct device *dev, struct device_driver *drv); /* 匹配操作 */
int (*uevent)(struct device *dev, struct kobj_uevent_env *env);
int (*probe)(struct device *dev); /* 探测操作 */
int (*remove)(struct device *dev); /* 移除操作 */
void (*shutdown)(struct device *dev); /* 关机操作 */
int (*suspend)(struct device *dev, pm_message_t state); /* 挂起操作 */
int (*resume)(struct device *dev); /* 恢复操作 */
struct bus_type_private *p; /* 指向内核内部使用的总线相关数据 */
/* 这里只写出了部分成员 */
};

```

其中 `struct bus_type_private` 类型的定义如下：

```

struct bus_type_private {
    struct kset subsys; /* 内核集合，用来代表这个总线 */
    struct kset *drivers_kset; /* 内核集合，用来容纳注册在总线上的驱动 */
    struct kset *devices_kset; /* 内核集合，用来容纳注册在总线上的设备 */
    unsigned int drivers_autoprobe:1; /* 是否自动探测的标志位 */
    struct bus_type *bus; /* 指向所联系的总线 */
    /* 这里只写出了部分成员 */
};

```

这里用了与 `struct device_driver` 类似的技巧将内部使用的数据域剥离了出来。

总线由 `struct bus_type_private` 的内核集成员 `subsys` 代表，但是这个集合并不直接收纳内核对象，而是由 `drivers_kset` 和 `devices_kset` 指向的内核集合分别来收纳驱动和设备。

总线的 `uevent` 操作会在 `devices` 子系统的 `uevent` 操作中被回调。也就是说，当有新的设备注册或者注销时，所属总线的 `uevent` 操作会被调用，因此可以在这里增加一些总线特有的环境变量。

总线注册时自动探测功能默认是打开的，可以通过它的 `drivers_autoprobe` 属性进行修改。

总线最重要的功能是执行设备与驱动的关联操作。关联操作包括以下几个步骤。

- step 1** 调用 `match` 函数对设备和驱动进行匹配操作。`match` 函数返回非 0 表示成功，0 表示失败。如果匹配失败则整个关联操作失败。
- step 2** 对匹配成功的设备调用总线的 `probe` 函数进行探测操作，如果总线没有 `probe` 函数，则调用相应驱动的 `probe` 函数。`probe` 函数返回 0 表示成功，负数为错误码。如果探测失败则整个关联操作失败。
- step 3** 将探测成功的设备和驱动绑定。

在探测之前引入一个匹配操作的主要目的是用简单的判断（如字符串比较等）方法快速确定一个驱动是否适合指定的设备，以避免将不适合的设备传给驱动做探测操作。

当总线上有设备或驱动注销时，将调用总线的 `remove` 函数进行移除操作，如果总线没有 `remove` 函数，则调用相应驱动的 `remove` 函数。移除操作必须与探测操作一一对应。

当系统关机或重启时，总线的 `shutdown` 操作将被调用。`suspend` 和 `resume` 操作则与系统的电源管理有关。当设备进入挂起状态时，调用 `suspend` 操作函数；当设备从挂起状态恢复时，调用 `resume` 操作函数。

一般来说,总线的 **match** 操作是必须实现的,但 **probe**, **remove**, **shutdown**, **suspend** 和 **resume** 等操作不需要实现,这些操作往往是由设备的驱动实现的。

15.2.4.2 总线的注册与注销

总线的注册与注销函数原型如下:

```
int bus_register(struct bus_type *bus); /* 注册总线 */
void bus_unregister(struct bus_type *bus); /* 注销总线 */
```

注册总线将使它出现在 **bus** 子系统中,并且还在它自己的目录中创建名称为 **drivers** 及 **devices** 的集合。

15.2.5 类别

class 子系统中收纳的都是各种类别。在类别中,设备是按功能而不是连接关系归类的,属于同一个类别的设备有着相近的功能。

很多设备并不属于某个总线,因此无法利用前述的“驱动——总线——设备”模式进行管理,但可以让它属于某个类别。实际上,凡是有具体功能、需要自动产生设备文件的设备必然属于某个类别。

15.2.5.1 类别数据类型

表示类别的数据类型定义如下:

```
struct class {
    const char *name; /* 类别的名称 */
    struct module *owner; /* 类别所属的模块 */
    int (*dev_uevent)(struct device *dev, struct kobj_uevent_env *env);
    void (*class_release)(struct class *class); /* 类别的释放操作 */
    void (*dev_release)(struct device *dev); /* 设备的释放操作 */
    int (*suspend)(struct device *dev, pm_message_t state); /* 挂起操作 */
    int (*resume)(struct device *dev); /* 恢复操作 */
    struct class_private *p; /* 指向内核内部使用的类别相关数据 */
    /* 这里只写出了部分成员 */
};
```

其中 **struct class_private** 类型的定义如下:

```
struct class_private {
    struct kset class_subsys; /* 代表类别的内核集合 */
    struct list_head class_interfaces; /* 链表头,用来链住类别上的各个接口 */
    struct class *class; /* 指向所联系的类别 */
    /* 这里只写出了部分成员 */
};
```

这里同样将内核内部使用的数据域剥离了出来。类别由 **struct class_private** 的内核集合成员 **class_subsys** 代表。

类别的 **dev_uevent** 操作会在 **devices** 子系统的 **uevent** 操作中被回调。也就是说,当有新

的设备注册或者注销时，所属类别的 `dev_uevent` 操作会被调用，因此可以在这里增加一些类别特有的环境变量。

`class_release` 操作是类别自己的释放操作，会在类别被注销时调用。

`dev_release` 操作则是给类别下的设备提供的释放操作。在设备释放时，如果设备和它的设备类型都没有提供 `release` 函数，则会调用它所属类别的 `dev_release` 函数。

类别的 `suspend` 和 `resume` 操作会在设备挂起或恢复时调用，通常这两个操作不需要实现，因为设备的驱动一般会提供挂起和恢复操作。

15.2.5.2 类别的注册与注销

下面是注册和注销类别的接口函数的原型：

```
int class_register(struct class *class); /* 注册类别 */
void class_unregister(struct class *class); /* 注销类别 */
```

注册类别将使它出现在 `class` 子系统中。

15.2.5.3 类别的创建与销毁

为了方便，内核还提供了动态创建并注册类别的接口函数，原型如下：

```
struct class *class_create(struct module *owner, const char *name);
```

其各个参数及返回值的含义解释如下。

- ◆ **owner**：类别的所属模块，一般为 `THIS_MODULE`。
- ◆ **name**：类别的名称。
- ◆ **返回值**：指向新创建的类别，创建或注册失败时为包装成指针的错误码。



当 `class_create` 函数创建设备失败时，返回值也不是 `NULL`。

与创建类别相对的操作是销毁类别，其接口函数原型如下：

```
void class_destroy(struct class *cls);
```

其中参数 `cls` 指向要销毁的类别。销毁操作必须和创建操作一一对应。

创建类别通常是为了在类别中动态创建新的设备。例如，可用下面的代码为第 13 章中 `kpad` 例程的字符设备创建类别以及设备：

```
/* 头文件 */
#include <linux/device.h>
/* 变量定义 */
struct class *cls;
struct device *dev;
/* 以下代码放在模块的初始化函数中 */
/* 创建设备模型中的类别 */
cls = class_create(THIS_MODULE, "kpad");
```

```

if (IS_ERR(cls)) {
    pr_debug("kpad_init: class_create ERR = %ld!\n", -PTR_ERR(cls));
    goto class_create_fail;
}
/* 创建设备模型中的设备 */
dev = device_create(cls, NULL, kpad.dev, "kpad");
if (IS_ERR(dev)) {
    pr_debug("kpad_init: device_create ERR = %ld!\n", -PTR_ERR(dev));
    goto device_create_fail;
}

```

这些代码将会在 `class` 子系统中创建一个新类别 `kpad`，然后在 `kpad` 类别下创建一个新设备 `kpad`。

在类别中创建设备的好处是可以让应用程序自动为设备创建设备文件。这个应用程序一般是 `udev`，它会监视内核对象发出的用户态事件，并根据 `/sys/class` 目录中的信息动态地创建和删除设备文件。因此，如果要使用 `udev`，则 `sysfs` 伪文件系统必须挂载在 `/sys` 目录下。

在嵌入式设备上，由 `busybox` 提供的 `mdev` 应用程序具有与 `udev` 类似的功能。

15.2.6 接口

接口是类别的一个从属数据类型，使用它可以对一些具有相同特征设备的注册和注销进行统一的管理。

15.2.6.1 接口数据类型

表示接口的数据类型定义如下：

```

struct class_interface {
    struct list_head node; /* 链表节点 */
    struct class *class; /* 所属类别 */
    int (*add_dev)(struct device *dev, struct class_interface *class_intf);
    void (*remove_dev)(struct device *dev, struct class_interface *class_intf);
};

```

接口有两个操作：添加设备操作 `add_dev` 和移除设备操作 `remove_dev`。当新设备注册时，将逐个调用所属类别上注册的接口的 `add_dev` 函数；当设备注销时，将逐个调用所属类别上注册的接口的 `remove_dev` 函数。

15.2.6.2 接口的注册与注销

接口的注册与注销函数原型如下：

```

int class_interface_register(struct class_interface *class_intf); /* 注册 */
void class_interface_unregister(struct class_interface *class_intf); /* 注销 */

```

注册接口将使它加入到所属类别的链表中，并且会对所属类别中已注册的设备逐个调用接口的 `add_dev` 函数；注销接口将把它从所属类别的链表中删除，并且会对所属类别中已注册的设备逐个调用接口的 `remove_dev` 函数。

从这里可以看出，接口与设备在类别中的关系类似于驱动与设备在总线中的关系，不管是新设备注册还是新接口注册，都要与类别中已注册的接口或设备一一进行关联操作。

15.3 常见总线与类别

内核中已经定义了很多总线和类别。下面将着重介绍其中的 **platform** 总线以及 **misc** 类别。

15.3.1 platform 总线

platform 总线实际上并不对应任何硬件上的总线，因此有时又称伪总线。由于设备模型中的驱动与设备关联机制必须要有一条总线才能发挥作用，对于那些没有连接在实际总线上的设备，如果想使用这一机制，就需要将它连接在一条假想的总线上。**platform** 总线就可以起到这个作用。通常，**platform** 总线上的设备都是直接与 CPU 连接的底层设备。

使用 **platform** 总线的好处是可以将驱动和设备分离，驱动所需的平台相关数据则在定义设备时提供，使驱动有更大的跨平台通用性。

platform 总线的相关定义与声明都在以下头文件中：

```
#include <linux/platform_device.h>
```

15.3.1.1 platform 总线基本特征

platform 总线类型的定义如下：

```
struct bus_type platform_bus_type = {
    .name      = "platform",
    .match      = platform_match,
    .uevent     = platform_uevent,
    /* 这里只写出了部分成员 */
};
```

可以看出，它的匹配操作函数是 **platform_match**，这个函数定义如下：

```
static int platform_match(struct device *dev, struct device_driver *drv)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct platform_driver *pdrv = to_platform_driver(drv);
    if (pdrv->id_table)
        return platform_match_id(pdrv->id_table, pdev) != NULL;
    return (strcmp(pdev->name, drv->name) == 0);
}
```

其中调用的 **platform_match_id** 函数定义如下：

```
static const struct platform_device_id *platform_match_id(
    struct platform_device_id *id, struct platform_device *pdev)
{
    while (id->name[0]) {
        if (strcmp(pdev->name, id->name) == 0) {
```



```

        pdev->id_entry = id;
        return id;
    }
    id++;
}
return NULL;
}

```

显然，在匹配的时候，先将设备的名称与驱动体的 `id_table` 成员数组中列举的名称逐一比较，如果相同则匹配成功；否则再将设备的名称与驱动的名称比较，如果相同则匹配成功。注意 `platform` 驱动的 `id_table` 是一个很新的特性，因此很少有驱动用到，而且这种匹配的机制有可能在将来版本的内核中改变。

`platform` 总线的用户态事件钩子函数是 `platform_uevent`，它的定义如下：

```

static int platform_uevent(struct device *dev, struct kobj_uevent_env *env)
{
    struct platform_device *pdev = to_platform_device(dev);
    add_uevent_var(env, "MODALIAS=%s%s", PLATFORM_MODULE_PREFIX,
        (pdev->id_entry) ? pdev->id_entry->name : pdev->name);
    return 0;
}

```

也就是说，`platform` 总线上的设备发送用户态事件时，会增加一个 `MODALIAS` 环境变量。

15.3.1.2 platform 设备

`platform` 总线将设备模型中的设备包装成了 `platform` 设备，其定义如下：

```

struct platform_device {
    const char *name; /* 设备名称 */
    int id; /* 设备编号，-1 表示只有一个 */
    struct device dev; /* 内嵌的设备对象 */
    u32 num_resources; /* 资源数组中元素的个数 */
    struct resource *resource; /* 指向描述资源的数组 */
    struct platform_device_id *id_entry; /* 保存与驱动匹配后的 ID */
};

```

设备最终的名称会被设为 `name.id`，其中 `id` 是设备的编号。如果 `id` 成员赋值为 `-1`，则表示这个设备只可能有一个，设备最终的名称就是 `name`。

`num_resources` 和 `resource` 成员共同描述设备要使用的全部资源，即端口号、IO 内存、中断号等，内核将它们统称为资源，并用统一的方法进行管理。表示资源的数据类型定义如下：

```

struct resource {
    resource_size_t start; /* 资源区域的起始值 */
    resource_size_t end; /* 资源区域的结束值 */
    const char *name; /* 申请资源的设备名称 */
    unsigned long flags; /* 资源的标志 */
    struct resource *parent, *sibling, *child; /* 树指针 */
};

```



每一个 `struct resource` 类型的数据描述资源的一段区域，同一类型的所有资源以树的形式组织在一起。资源的标志包含了资源类型的说明，常用的几个类型如下。

- ◆ `IORESOURCE_IO`: 端口号资源。
- ◆ `IORESOURCE_MEM`: IO 内存资源。
- ◆ `IORESOURCE_IRQ`: 中断号资源。
- ◆ `IORESOURCE_DMA`: DMA 资源。

平台相关的其他数据则放在 `platform` 设备的 `dev` 成员的 `platform_data` 指针所指向的内存中。下面是 `platform` 设备的注册与注销操作函数的原型：

```
int platform_device_register(struct platform_device *pdev); /* 注册 */
void platform_device_unregister(struct platform_device *pdev); /* 注销 */
```

注册 `platform` 设备时，首先向 `platform` 总线注册相应的设备，然后将设备的端口号和 IO 内存资源添加到系统的资源树中。注销 `platform` 设备则是相反的操作。

通常一个系统中的 `platform` 设备的个数和类型总是固定的，因此可以把它们的地址放在一个数组中，然后用下面的接口函数同时注册：

```
int platform_add_devices(struct platform_device **devs, int num);
```

其各个参数及返回值的含义如下。

- ◆ `devs`: 指向一个数组，数组的每个元素指向一个 `platform` 设备。
- ◆ `num`: 数组中元素的个数（设备的个数）。
- ◆ 返回值：成功返回 0，负数表示错误码。

15.3.1.3 platform 驱动

`platform` 总线将设备模型中的驱动包装成了 `platform` 驱动，其定义如下：

```
struct platform_driver {
    int (*probe)(struct platform_device *pdev); /* 探测操作 */
    int (*remove)(struct platform_device *pdev); /* 移除操作 */
    struct device_driver driver; /* 内嵌的驱动对象 */
    struct platform_device_id *id_table; /* 用于匹配的 ID 数组 */
    /* 这里只写出了部分成员 */
};
```

在 `platform` 驱动的成员 `driver` 的各种操作函数中，实际上直接回调了 `platform` 驱动的各种操作，只是将参数的类型转换了一下。因为 `platform` 总线本身没有 `probe` 操作和 `remove` 操作，所以当 `platform` 驱动注册时，如果 `platform` 总线上已经注册了匹配的设备，就会调用驱动的 `probe` 方法。



`platform` 设备通常先于 `platform` 驱动而加载，这样探测操作就会只发生在模块的加载过程中，从而可以安全地放入初始化数据段，以节约内存。

注册和注销 platform 驱动接口函数原型如下：

```
int platform_driver_register(struct platform_driver *drv); /* 注册 */
void platform_driver_unregister(struct platform_driver *drv); /* 注销 */
```

在 platform 驱动的 probe 函数中，可以用下面这个函数获得设备的各种资源：

```
struct resource *platform_get_resource(struct platform_device *dev,
    unsigned int type, unsigned int num);
```

其各个参数及返回值的含义解释如下。

- ◆ dev：指向要获得资源的 platform 设备。
- ◆ type：资源的类型。
- ◆ num：资源的索引，表示要获得设备的资源数组中第几个此类型的资源，索引从 0 开始编号。
- ◆ 返回值：指向获得的资源，NULL 表示没有这个资源。

如果要获得中断号资源，还可以用下面这个函数：

```
int platform_get_irq(struct platform_device *dev, unsigned int num);
```

其各个参数及返回值的含义解释如下。

- ◆ dev：指向要获得中断资源的 platform 设备。
- ◆ num：资源的索引，表示要获得设备的资源数组中第几个中断资源，索引从 0 开始编号。
- ◆ 返回值：所获得的中断号，负数表示错误码。

获得的中断号必须经过申请才能使用。

15.3.2 misc 类别

如果驱动要注册字符设备，则可以先创建一个新类别，然后在新类别中为它创建对应的设备对象。事实上，对于单个或少量的字符设备，完全可以用直接注册 misc 设备的方式来处理。

misc 是内核在启动阶段注册的一个类别。注册 misc 类别的同时，内核注册了一个字符设备，占据了主设备号 MISC_MAJOR 下 0~255 范围内的次设备号。内核的主设备号定义在头文件 <linux/major.h> 中，如：

```
#define MISC_MAJOR 10
```

这个字符设备只支持 open 操作，在 open 操作中再将文件操作替换为已注册的 misc 设备的文件操作。由于这时还没有向 class 子系统注册设备，因此这个字符设备的设备文件不会被自动创建。

向 misc 类别注册设备的接口在以下头文件中：

```
#include <linux/miscdevice.h>
```

表示 misc 设备的数据类型定义如下：

```

struct miscdevice {
    int minor; /* 次设备号 */
    const char *name; /* 设备名称 */
    const struct file_operations *fops; /* 设备的文件操作 */
    struct list_head list; /* 链表节点, 用来加入 misc 设备队列 */
    struct device *parent; /* 指向父设备 */
    struct device *this_device; /* 指向本设备 */
};

```

其各个成员的含义解释如下。

- ◆ **minor**: 指定次设备号, 如果设置为 `MISC_DYNAMIC_MINOR`, 则注册时会自动为设备分配一个次设备号。
- ◆ **name**: 设备的名称。
- ◆ **fops**: 设备的文件操作。
- ◆ **list**: 链表节点, 用来加入 `misc` 设备队列。
- ◆ **parent**: 指向设备的父设备, 注意它是设备模型中的设备对象。
- ◆ **this_device**: 指向为本设备创建的设备模型中的设备对象。

`misc` 设备的注册与注销函数的原型如下:

```

int misc_register(struct miscdevice *misc); /* 注册 misc 设备 */
int misc_deregister(struct miscdevice *misc); /* 注销 misc 设备 */

```

其中参数 `misc` 指向要注册或注销的 `misc` 设备, 返回值为 0 表示操作成功, 负数表示错误码。

注册 `misc` 设备时, 一方面是向 `misc` 类别注册了新设备, 因此设备文件会被自动创建; 另一方面, 对应设备号下的文件操作会被替换为 `misc` 设备的文件操作, 因此实现了新设备的功能。



第 16 章 Linux 驱动实例详解

内核所支持的设备类型非常繁杂，它们的编程接口也各不相同。本章主要以两类驱动的实际例子来说明实际的设备驱动是如何编写的：一类为输入设备驱动，另一类为 USB 驱动。

输入设备驱动以适用于 HY2410A 开发板的触摸屏驱动为例，这是一个实际可以使用的驱动；USB 驱动以内核的 `usb-skeleton` 驱动为范例，只是由笔者重新加了注释，这个驱动虽然不是针对任何具体的设备编写的，但也可以编译通过且有一定的功能。

对驱动的介绍都采用先介绍编程接口、后给出驱动例程的方式。

16.1 输入设备驱动

输入设备驱动是 Linux 中常见的一大类设备驱动。由于嵌入式系统经常会定制自己的输入设备，这些设备的驱动不可能都包含在标准的内核源码中，只能自己开发。所以掌握输入设备驱动的编程有着较大的实践意义。

Linux 内核的输入体系的架构如图 16.1 所示。输入设备硬件可能直接连入系统，也可能经由其他总线接入，因此输入设备驱动可能直接面向硬件，也可能面向下层的总线驱动。输入设备驱动负责管理具体的输入设备硬件，其最主要的任务就是注册输入设备，然后读取硬件产生的各种信号，并转化为输入事件，向输入系统的核心报告。

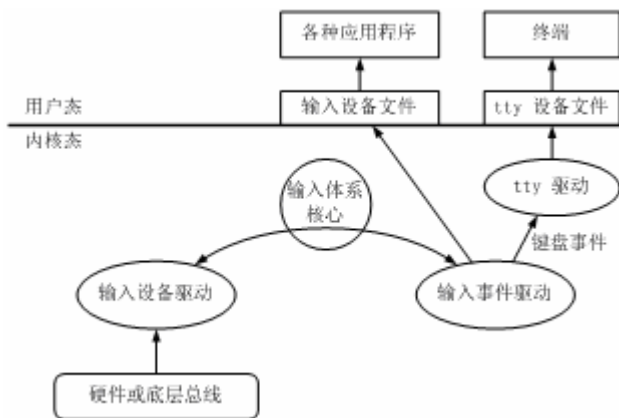


图 16.1 输入体系结构

在系统启动时，输入体系的核心在 `class` 子系统内注册一个类别 `input`，并且注册主设备号为 `INPUT_MAJOR`、次设备号为 `0-255` 的字符设备。`INPUT_MAJOR` 的定义如下：

```
#define INPUT_MAJOR 13
```

与 `misc` 类别相似，这时并没有向 `class` 子系统注册设备对象，因此不会自动出现 255 个字

符设备。

输入设备驱动利用输入体系核心提供的接口向 **input** 类别注册输入设备，但是所注册的设备并不对应着任何字符设备，也就是说没有对应的设备文件。因此输入设备驱动报告的事件不能直接到达应用程序，其间还要经过输入事件驱动的过滤和处理。

输入事件驱动与具体的硬件设备无关，只关心设备所报告的事件。经过处理的事件以设备文件的形式向应用程序传输。特殊的是键盘事件，它可以直接向内核的 **tty** 驱动传输，终端程序打开的是 **tty** 设备文件而不是输入设备文件。

输入设备与事件驱动的关系类似于总线上的设备与驱动之间的关系，每当一个新的输入设备注册，或者一个新的事件驱动注册时，由输入体系的核心代码进行设备与驱动的关联操作。关联操作分两个阶段，首先调用事件驱动的匹配操作，匹配操作仅根据输入设备的基本参数进行简单判断以决定能否关联；匹配成功后调用事件驱动的连接操作，进行事件设备的初始化和注册工作。事件设备也注册在 **input** 类别内，并且对应着输入体系核心所注册的字符设备。

输入设备与事件驱动之间是多对多的关系。同一个输入设备报告的事件可能被多个事件驱动处理，并以不同的形式通知应用程序。

Linux 2.6.30 内核源码中包含了两个基本的事件驱动：**evdev** 驱动和 **mousedev** 驱动，其中 **evdev** 驱动有以下特点。

- ◆ 对事件的过滤和处理操作最少，基本上将设备报告的事件以原始形式提供给应用程序。
- ◆ 与各个输入设备无条件匹配。
- ◆ 对应于每个关联的输入设备，创建一个名为 **event%** 的事件设备，% 是一个编号。

而 **mousedev** 驱动则与鼠标有关，它有如下特点。

- ◆ 对事件进行处理，以鼠标事件的格式上报事件。
- ◆ 输入设备必须能够产生坐标事件才能匹配。
- ◆ 对应于每个关联的输入设备，创建一个名为 **mouse%** 的事件设备，% 是一个编号。
- ◆ 无条件创建事件设备 **mice**，所有关联的输入设备报告的事件都通过它反映出来。

因为内核中有了这些事件驱动，所以实际应用中一般只需要编写输入设备驱动即可。

16.1.1 输入设备编程接口

下面将对输入设备驱动的编程接口进行介绍。相关的定义和声明都在以下头文件中：

```
#include <linux/input.h>
```

16.1.1.1 输入设备数据类型

输入设备由 **struct input_dev** 数据类型代表，它的定义如下：

```
struct input_dev {
    const char *name; /* 设备名称 */
    const char *phys; /* 设备的物理路径 */
    const char *uniq; /* 设备的独特标识码 */
    struct input_id id; /* 设备标识，包含总线类型、制造商标识、产品标识、版本号 */
};
```

```

/* 以上内容与功能无关 */
unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; /* 事件类型使能标志 */
unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; /* 按键事件使能标志 */
unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; /* 相对坐标事件使能标志 */
unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; /* 绝对坐标事件使能标志 */
unsigned int keycodemax; /* 键盘码表项的个数 */
unsigned int keycodesize; /* 键盘码表项的大小 */
void *keycode; /* 键盘码表的首地址 */
int (*setkeycode)(struct input_dev *dev, int scancode, int keycode);
int (*getkeycode)(struct input_dev *dev, int scancode, int *keycode);
int absmax[ABS_MAX + 1]; /* 绝对坐标最大值 */
int absmin[ABS_MAX + 1]; /* 绝对坐标最小值 */
int absfuzz[ABS_MAX + 1]; /* 绝对坐标噪声 */
int absflat[ABS_MAX + 1]; /* 绝对坐标的中央平坦区域，用于游戏杆 */
int (*open)(struct input_dev *dev); /* 打开操作 */
void (*close)(struct input_dev *dev); /* 关闭操作 */
struct device dev; /* 设备对象 */
/* 这里只写出了部分成员 */
};

```

其中的 `name`、`phys` 等成员与设备的功能无关，但是可以被应用程序通过 `ioctl` 系统调用查询，并且也会出现在设备对象的属性上，也就是可以通过 `sysfs` 文件系统中的相应目录下的文件查询。

`input_dev` 结构体有很多名为 `xxxbit` 的成员，这些成员以标志位的方式表示设备的功能，其中成员 `evbit` 的每个比特位都表示设备对于某种事件类型是否支持，而其他成员的每个比特位都表示对某类型事件中具体某个事件是否支持。这些标志位必须正确设置。首先，与事件驱动的匹配主要就依据这些标志位；其次，如果设备报告了它不支持的事件，事件将直接被过滤掉。

成员 `keycodemax`、`keycodesize` 和 `keycode` 用于保存键盘码表的信息。这里码表是一个一般化的数组，它的每个元素的大小是 `keycodesize`，数组中元素的个数是 `keycodemax`，而数组的首地址则保存在 `keycode` 里。码表用来进行从扫描码到键盘码的转换。键盘设备驱动从硬件直接读到的编码称为扫描码，以扫描码为下标取码表元素的值，得到的应是键盘码，键盘码是报告事件时使用的编码。显然，扫描码与键盘码的对应关系完全由设备驱动所决定。实际上，输入体系核心及事件驱动都不会直接访问这些成员，因此它们基本可以作为设备驱动的私有数据使用。

`setkeycode` 和 `getkeycode` 两个操作用于支持应用程序通过 `ioctl` 系统调用查询码表和设置码表的功能，可以不实现。

`open` 和 `close` 是设备的打开和关闭操作。当应用程序打开设备文件时，首先进入输入事件驱动的打开操作，在其中一般会回调输入设备的打开操作。关闭操作与此类似。

代表输入设备的结构体只能由以下函数进行动态分配：

```
struct input_dev *input_allocate_device(void);
```

这个函数的返回值指向分配到的输入设备，如果分配失败，则返回 `NULL`。

与这个函数对应的是释放输入设备的函数：

```
void input_free_device(struct input_dev *dev);
```

其中参数 `dev` 指向要释放的输入设备。

需要注意的是，输入设备如果注册成功，则在注销的时候会自动释放，因此这个函数只能用在注册失败的情况下。

16.1.1.2 设置输入设备的功能

由 `input_allocate_device` 函数分配得到的输入设备已经进行了基本的初始化，但还有很多数据域需要根据设备的特性来进行具体的设置，其中最重要的步骤就是设置设备的功能。输入体系核心提供了一个设置功能的函数，原型如下：

```
void input_set_capability(struct input_dev *dev,
    unsigned int type, unsigned int code);
```

其各个参数的含义解释如下。

- ◆ `dev`: 指向要设置的输入设备。
- ◆ `type`: 要设置的事件类型。
- ◆ `code`: 要设置的事件的编码。

常用的事件类型有以下几类。

- ◆ `EV_KEY`: 表示按键事件，按键事件不仅仅指键盘的按键，也包括鼠标的左右键、触摸屏的触摸等事件。
- ◆ `EV_REL`: 相对坐标事件，通常用于鼠标。
- ◆ `EV_ABS`: 绝对坐标事件，通常用于触摸屏。

事件的编码根据事件的类型而有所不同，当类型为 `EV_KEY` 时，事件的编码一般是形如 `KEY_xxx` 或 `BTN_xxx` 的宏，如键盘的 A 键是 `KEY_A`，而鼠标的左键是 `BTN_LEFT`；当类型为 `EV_REL` 时，通常是 `REL_X`、`REL_Y` 和 `REL_Z`，分别代表三个方向的坐标事件；当类型为 `EV_ABS` 时，通常是 `ABS_X`、`ABS_Y` 和 `ABS_Z`，也分别代表三个方向的坐标事件。所有的事件定义可在 `<linux/input.h>` 头文件中找到。

实际上这个函数就是在设置输入设备的各个标志位。为了简单，有时候也会直接设置这些标志位，如：

```
struct input_dev *dev = input_allocate_device(void); /* 分配输入设备 */
dev->evbit[BIT_WORD(EV_KEY)] |= BIT_MASK(EV_KEY); /* 设置 EV_KEY 标志 */
```

由于标志位的数量多数都超过 32 位，不能放在一个长整型数据内，所以被定义成一个长整型的数组。这里，`BIT_WORD` 用来得到标志位所在的数组元素下标，而 `BIT_MASK` 则用来构造一个长整型数据，这个数据只有对应的标志位上是 1，其他比特位全部是 0。

用 `set_bit` 函数则更简单，如：

```
set_bit(EV_KEY, dev->evbit); /* 设置 EV_KEY 标志 */
```

对于支持绝对坐标的设备来说，还有一些属性要设置，可以利用以下函数：

```
void input_set_abs_params(struct input_dev *dev,
                        int axis, int min, int max, int fuzz, int flat);
```

其各个参数的含义解释如下。

- ◆ dev: 指向要设置的输入设备。
- ◆ axis: 坐标轴的方向, 可选值为 ABS_X, ABS_Y, ABS_Z 等。
- ◆ min: 最小值。
- ◆ max: 最大值。
- ◆ fuzz: 噪声。
- ◆ flat: 中央平坦区。

16.1.1.3 输入设备的注册与注销

初始化好的输入设备可以向 input 类别注册, 其接口函数原型如下:

```
int input_register_device(struct input_dev *dev);
```

这里参数 dev 指向要注册的输入设备, 返回值为 0 表示注册成功, 如果为负数, 则代表一个错误码。

相反的操作是注销输入设备, 其接口函数原型如下:

```
void input_unregister_device(struct input_dev *dev);
```

其中参数 dev 指向要注销的输入设备。注销时, 代表输入设备的结构体所占的内存空间将被释放。

16.1.1.4 报告事件

输入设备驱动最重要的任务就是报告各种事件。当驱动检测到硬件上发生信号时, 如某个按键被按下或松开, 就要报告相应的事件。报告按键事件使用以下函数:

```
void input_report_key(struct input_dev *dev, unsigned int code, int value);
```

其各个参数的含义解释如下。

- ◆ dev: 指向要报告事件的输入设备。
- ◆ code: 事件的编码。
- ◆ value: 事件的值, 1 表示键按下, 0 表示键松开。

报告相对坐标事件使用以下函数:

```
void input_report_rel(struct input_dev *dev, unsigned int code, int value);
```

其各个参数的含义解释如下。

- ◆ dev: 指向要报告事件的输入设备。
- ◆ code: 事件的编码。
- ◆ value: 事件的值, 即相对坐标的值。

报告绝对坐标事件使用以下函数：

```
void input_report_abs(struct input_dev *dev, unsigned int code, int value);
```

其各个参数的含义解释如下。

- ◆ dev: 指向要报告事件的输入设备。
- ◆ code: 事件的编码。
- ◆ value: 事件的值, 即绝对坐标的值。

事件报告以后将被存放在队列里, 只有再报告一个同步事件以后, 队列中的事件才会得到处理。报告同步事件使用以下函数：

```
void input_sync(struct input_dev *dev);
```

其中参数 dev 指向要报告事件的输入设备。

16.1.2 触摸屏驱动例程

下面将以 HY2410A 开发板为例进行触摸屏驱动的设计。HY2410A 采用 S3C2410A 作为主控制器, 它支持常用的四线电阻式触摸屏。

16.1.2.1 触摸屏工作原理

简单地说, 电阻式触摸屏由两层叠在一起的薄膜组成, 它们靠中间的一侧都涂有带一定阻抗的导电涂层, 而两层中间填充有弹性材料作为支撑。在自然状态下, 两边的导电涂层是不接触的, 当某处受到一定压力时, 就会导致两边的导电涂层在这一点上发生接触, 电路导通。从中间某点到触摸屏边缘的电阻基本与它到边缘的距离成正比, 如果能够检测这个电阻的相对大小, 就可以确定点的坐标。

在其中一层薄膜的上下两侧分别引出一对电极, 称为 YP 和 YM, 在另一层薄膜的左右两侧分别引出一对电极, 称为 XP 和 XM, 这样就构成了四线电阻式触摸屏。它的工作原理可由图 16.2 解释。在自然状态下, 电极 YP 与 YM 相连, XP 与 XM 相连。当某一点被按下时, 四个电极相互连在一起, 这一点相当于通过四个等效的电阻连在四个电极上。

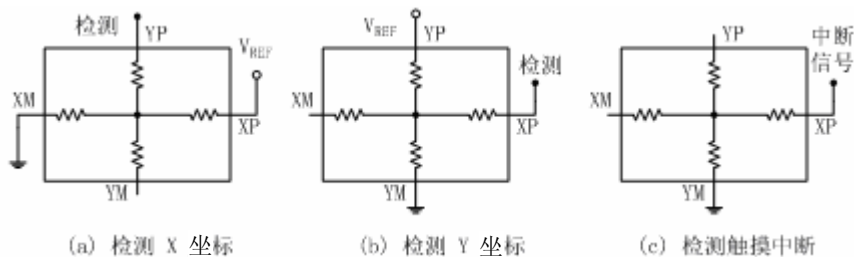


图 16.2 四线电阻式触摸屏工作原理

要检测 X 方向坐标时, 如图 16.2 (a) 所示, 将 XP 电极接参考电压 V_{REF} , XM 接地, YM 断开, 这时从 YP 检测到的电压值相当于 X 方向两个电阻对 V_{REF} 的分压, 它正比于 XM 端电阻

的大小，根据这个电压就可以确定接触点在 X 方向的坐标。

反之，如果将 YP 电极接参考电压 VREF，YM 接地，XM 断开，则根据从 XP 检测到的电压值就可以确定接触点在 Y 方向的坐标，如图 16.2 (b) 所示。

当触摸屏在自然状态时，如果将电极 YP 和 XM 断开，YM 接地，那么无论哪一点被按下，都将导致 XP 与 YM 通过两个电阻连接在一起。如果 XP 端事先通过一个大电阻上拉到正电压，那么这种连接将使 XP 端检测到的电压下降，这可以用来作为触摸事件发生的中断信号。

通过检测端得到的电压值是模拟信号，需要经过模数转换变为数字信号。S3C2410 芯片集成了带 8 路信号选择器的 ADC，其中的 AIN[7] 与 AIN[5] 管脚专门用来进行触摸屏的检测，它与触摸屏的电路连接如图 16.3 所示。

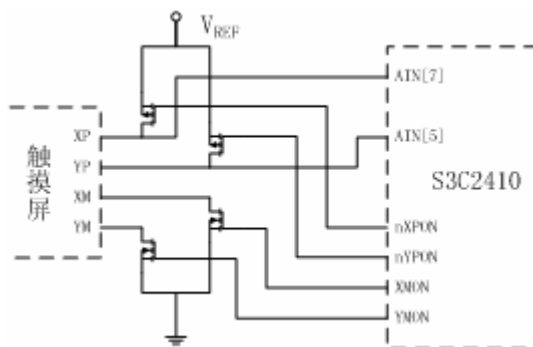


图 16.3 S3C2410 触摸屏电路示意图

触摸屏的 XP 和 YP 电极经过 P 沟道增强型场效应管接到参考电压，场效应管的栅极由 S3C2410 的 nXPON 和 nYPON 信号控制。当这两个信号为低电平时，场效应管导通，相当于 YP 或 XP 接到了参考电压；当这两个信号为高电平时，场效应管截止，YP 或 XP 只与 AIN[7] 和 AIN[5] 相连。

触摸屏的 XM 和 YM 电极经过 N 沟道增强型场效应管接地，场效应管的栅极由 XMON 和 YMON 信号控制。当这两个信号为低电平时，场效应管截止，相当于 XM 或 YM 悬空；当这两个信号为高电平时，场效应管导通，相当于 XM 或 YM 接地。

正确控制这些信号就可以从 AIN[7] 和 AIN[5] 管脚上检测触摸点的坐标。S3C2410 内部对 AIN[7] 输入的信号有上拉功能，并且根据 AIN[7] 的信号可以产生中断，用来表示发生了触摸事件。

16.1.2.2 触摸屏的硬件支持

S3C2410 的 nXPON, nYPON, XMON 和 YMON 四个管脚实质上都是 GPIO 管脚，是与其他功能复用的，因此在控制触摸屏之前这四个管脚必须配置为正确的功能。Linux 2.6.30 内核对 S3C2410 已经有了充分的支持，配置 GPIO 管脚的功能可以使用下面这个函数：

```
void s3c2410_gpio_cfgpin(unsigned int pin, unsigned int function);
```

其中参数 pin 是管脚的配置地址，而 function 是一个功能代码，它们的取值都在以下头文件中进行了定义：

```
#include <mach/regs-gpio.h>
```

直接使用其中的宏，可以省去查阅芯片手册并进行推算的麻烦。

对 ADC 的控制使用内存映射 IO 的方式，相关的定义在下面这个头文件中：

```
#include <plat/regs-adc.h>
```

由于 ADC 集成在 S3C2410 芯片内部，因此它的物理地址是固定的，各个寄存器被映射在从物理地址 S3C24XX_PA_ADC 开始、长度为 S3C24XX_SZ_ADC 的内存中。将这个地址映射为虚拟地址，再使用上述头文件中定义的寄存器地址偏移量，我们就可以访问到这些寄存器，通过这些寄存器可以控制 ADC 的以下功能。

- ◆ 转换时钟，S3C2410 的 ADC 是逐次逼近型，需要多个时钟周期才能完成一次转换。设置时钟频率将影响 ADC 的转换速度。
- ◆ 工作模式切换，ADC 有不同的工作模式，可以通过写寄存器进行切换。
- ◆ 启动转换，可以通过写寄存器的方式启动 ADC 的转换。
- ◆ 检测铁笔是否触屏，通过数据寄存器中的标志位可以进行检测。
- ◆ 得到转换结果，通过读数据寄存器的方式。

由 S3C2410 的芯片手册可知，它支持一种自动 XY 模式的转换。在这种模式下，当启动 ADC 进行转换时，硬件自动控制 nXPON, nYPON, XMON 和 YMON 信号，先检测 X 方向坐标，再检测 Y 方向坐标，检测的结果分别放在 ADCDAT0 和 ADCDAT1 寄存器中。两个方向的坐标全部采样完毕后触发 IRQ_ADC 中断。这是最方便的驱动触摸屏的方式。

当 ADC 被设置为等待中断模式时，一旦有铁笔触屏，就会触发 IRQ_TC 中断。

16.1.2.3 触摸屏驱动设计

根据以上对硬件支持的分析，可以初步决定软件的设计方案。对于一个触摸屏设备来说，至少要能够上报以下三种事件。

- ◆ BTN_TOUCH：铁笔的触屏事件，当铁笔接触或离开屏时要上报。
- ◆ ABS_X：X 方向绝对坐标，在铁笔触屏状态下上报。
- ◆ ABS_Y：Y 方向绝对坐标，在铁笔触屏状态下上报。

铁笔触屏的事件可以通过 IRQ_TC 检测，但离屏时没有对应的中断可用，因此需要在铁笔触屏后进入查询状态，通过周期性地查询铁笔状态来判断铁笔何时离屏。这里要用到一个定时器。同时，在铁笔触屏的状态下，需要不断地启动 ADC 进行转换并上报坐标事件，这里也要用到一个定时器。为了简单且避免同步问题，两个定时器可以合二为一。

ADC 采样结束后将触发 IRQ_ADC 中断，因此可以在中断处理函数中读取采样后的数据。

16.1.2.4 触摸屏驱动源码

HY2410A 触摸屏驱动的源码如下：

```
/* 文件名: hy2410_ts.c */
/* 说明: HY2410 触摸屏驱动例程 */
```

```

#include <linux/module.h>
#include <linux/input.h> /* 输入设备 */
#include <linux/interrupt.h> /* 中断 */
#include <linux/ioport.h> /* IO 内存申请 */
#include <linux/io.h> /* IO 内存读写 */
#include <linux/timer.h> /* 定时器 */
#include <linux/irq.h> /* 中断定义 */
#include <linux/clk.h> /* 时钟 */
#include <plat/regs-adc.h> /* ADC 寄存器定义 */
#include <mach/regs-gpio.h> /* GPIO 管脚定义 */

MODULE_LICENSE("GPL");

/* 设备名称 */
#define DEVNAME "hy2410-touchscreen"

/* ADC 参数 */
#define ADC_PRESCALER 49 /* 时钟预分频值 */
#define ADC_DELAY 10000 /* 转换延时 */
#define NR_SAMPLING 4 /* 采样次数, 最好是 2 的整数次幂 */

/* 绝对坐标的最大最小值 */
#define X_AXIS_MAX 0x3ff
#define X_AXIS_MIN 0
#define Y_AXIS_MAX 0x3ff
#define Y_AXIS_MIN 0

/* 设备私有数据 */
struct ts_data {
    void __iomem *iobase; /* IO 内存虚拟地址 */
    struct clk *clk; /* 时钟源 */
    unsigned int x; /* X 坐标 */
    unsigned int y; /* Y 坐标 */
    int count; /* 采样次数 */
    struct timer_list timer; /* 定时器 */
};

/* 读 ADC 数据寄存器 0 */
#define ADCDAT0(ts) ioread32((ts)->iobase+S3C2410_ADCCDAT0)
/* 读 ADC 数据寄存器 1 */
#define ADCDAT1(ts) ioread32((ts)->iobase+S3C2410_ADCCDAT1)
/* 读 ADC 控制寄存器 */
#define ADCCON(ts) ioread32((ts)->iobase+S3C2410_ADCCON)
/* 写 ADC 控制寄存器 */
#define wADCCON(ts, data) iowrite32(data, (ts)->iobase+S3C2410_ADCCON)
/* 读 ADC 触摸屏控制寄存器 */
#define ADCTSC(ts) ioread32((ts)->iobase+S3C2410_ADCTSC)
/* 写 ADC 触摸屏控制寄存器 */
#define wADCTSC(ts, data) iowrite32(data, (ts)->iobase+S3C2410_ADCTSC)
/* 读 ADC 转换延时寄存器 */
#define ADCDLY(ts) ioread32((ts)->iobase+S3C2410_ADCDLY)

```



```

/* 写 ADC 转换延时寄存器 */
#define wADCDLY(ts, data) iowrite32(data, (ts)->iobase+S3C2410_ADCDLY)

/* 设置 ADC 时钟的预分频值 */
static inline void set_adc_prescaler(struct ts_data *ts, int value)
{
    unsigned int data;
    data = ADCCON(ts); /* 读原值 */
    data |= S3C2410_ADCCON_PRSCEN; /* 设置预分频使能标志位 */
    data &= ~S3C2410_ADCCON_PRSCVLMASK; /* 清空原来的预分频值 */
    data |= S3C2410_ADCCON_PRSCVL(value); /* 设置新的预分频值 */
    wADCCON(ts, data); /* 写新值 */
}

/* 设置 ADC 的备用模式 */
static inline void set_adc_standby(struct ts_data *ts, int on)
{
    unsigned int data;
    data = ADCCON(ts); /* 读原值 */
    if (on) {
        data |= S3C2410_ADCCON_STDBM; /* 设置备用模式标志位 */
    } else {
        data &= ~S3C2410_ADCCON_STDBM; /* 清除备用模式标志位 */
    }
    wADCCON(ts, data); /* 写新值 */
}

/* 设置 ADC 的读触发模式 */
static inline void set_adc_read_start(struct ts_data *ts, int on)
{
    unsigned int data;
    data = ADCCON(ts); /* 读原值 */
    if (on) {
        data |= S3C2410_ADCCON_READ_START; /* 设置读触发标志位 */
    } else {
        data &= ~S3C2410_ADCCON_READ_START; /* 清除读触发标志位 */
    }
    wADCCON(ts, data); /* 写新值 */
}

/* 设置 ADC 的转换延时 */
static inline void set_adc_delay(struct ts_data *ts, int value)
{
    wADCDLY(ts, value);
}

/* 启动 ADC 进行转换 */
static inline void start_adc(struct ts_data *ts)
{
    unsigned int data;
    data = ADCCON(ts); /* 读原值 */

```

```

    data |= S3C2410_ADCCON_ENABLE_START; /* 设置启动标志位 */
    wADCCON(ts, data); /* 写新值 */
}

/* 设置 ADC 为自动 XY 模式 */
static inline void set_adc_autoxy(struct ts_data *ts)
{
    unsigned int data;
    data = ADCTSC(ts); /* 读原值 */
    data |= S3C2410_ADCTSC_PULL_UP_DISABLE; /* 设置 XP 禁用上拉标志位 */
    data |= S3C2410_ADCTSC_AUTO_PST; /* 设置自动 XY 模式标志位 */
    data &= ~S3C2410_ADCTSC_XY_PST(3); /* 清除等待中断模式标志位 */
    wADCTSC(ts, data); /* 写新值 */
}

/* 设置 ADC 为等待中断模式 */
static inline void set_adc_wait4int(struct ts_data *ts)
{
    unsigned int data;
    data = ADCTSC(ts); /* 读原值 */
    data &= ~S3C2410_ADCTSC_PULL_UP_DISABLE; /* 清除 XP 禁用上拉标志位 */
    data |= S3C2410_ADCTSC_YM_SEN; /* 设置标志位使 YM 接地 */
    data |= S3C2410_ADCTSC_YP_SEN; /* 设置标志位使 YP 接 AIN5 */
    data &= ~S3C2410_ADCTSC_XM_SEN; /* 清除标志位使 XM 为高阻 */
    data |= S3C2410_ADCTSC_XP_SEN; /* 设置标志位使 XP 接 AIN7 */
    data |= S3C2410_ADCTSC_XY_PST(3); /* 增加等待中断模式标志位 */
    data &= ~S3C2410_ADCTSC_AUTO_PST; /* 清除自动 XY 模式标志位 */
    wADCTSC(ts, data); /* 写新值 */
}

/* 判断铁笔是否触屏 */
static inline int is_stylus_on(struct ts_data *ts)
{
    unsigned int data0, data1;
    data0 = ADCDAT0(ts); /* 读数据寄存器 0 */
    data1 = ADCDAT1(ts); /* 读数据寄存器 1 */
    return !((data0 & S3C2410_ADCDAT0_UPDOWN) ||
            (data1 & S3C2410_ADCDAT1_UPDOWN));
}

/* 清空坐标数据 */
static inline void clear_data(struct ts_data *ts)
{
    ts->x = 0;
    ts->y = 0;
    ts->count = 0;
}

/* 定时器函数 */
static void ts_timer(unsigned long data)
{

```

```

struct input_dev *dev = (struct input_dev *)data; /* 得到输入设备 */
struct ts_data *ts = input_get_drvdata(dev); /* 得到私有数据 */
set_adc_wait4int(ts); /* 只有在等待中断模式下才能正确查询铁笔状态 */
if (is_stylus_on(ts)) { /* 铁笔触屏 */
    if (ts->count != 0) { /* 已有数据 */
        ts->x /= NR_SAMPLING; /* 求 X 坐标平均值 */
        ts->y /= NR_SAMPLING; /* 求 Y 坐标平均值 */
        input_report_abs(dev, ABS_X, ts->x); /* 报告 X 坐标事件 */
        input_report_abs(dev, ABS_Y, ts->y); /* 报告 Y 坐标事件 */
        clear_data(ts); /* 清空坐标数据 */
    } else { /* 无数据, 说明铁笔刚触屏, ADC 尚未转换 */
        input_report_key(dev, BTN_TOUCH, 1); /* 报告触摸事件发生 */
    }
    input_sync(dev); /* 同步事件 */
    /* 启动 ADC 转换 */
    set_adc_autoxy(ts);
    start_adc(ts);
} else { /* 铁笔离屏 */
    clear_data(ts); /* 清空数据 */
    input_report_key(dev, BTN_TOUCH, 0); /* 报告触摸事件消失 */
    input_sync(dev); /* 同步事件 */
    enable_irq(IRQ_TC); /* 使能中断 */
}
}

/* 中断 IRQ_TC 处理函数 */
static irqreturn_t tc_handler(int irq, void *dev_id)
{
    struct input_dev *dev = (struct input_dev *)dev_id; /* 得到输入设备 */
    struct ts_data *ts = input_get_drvdata(dev); /* 得到私有数据 */
    /* 铁笔未触屏, 视为干扰, 不处理 */
    if (!is_stylus_on(ts)) return IRQ_HANDLED;
    pr_debug("tc_handler: be called.\n");
    disable_irq_nosync(IRQ_TC); /* 进入查询模式, 禁止中断 */
    mod_timer(&ts->timer, jiffies+HZ/100); /* 启动定时器 */
    return IRQ_HANDLED;
}

/* 中断 IRQ_ADC 处理函数 */
static irqreturn_t adc_handler(int irq, void *dev_id)
{
    struct input_dev *dev = (struct input_dev *)dev_id; /* 得到输入设备 */
    struct ts_data *ts = input_get_drvdata(dev); /* 得到私有数据 */
    pr_debug("adc_handler: be called, %d time sampling.\n", ts->count);
    ts->x += ADCDAT0(ts) & S3C2410_ADCDAT0_XPDATA_MASK; /* 读 X 坐标并累加 */
    ts->y += ADCDAT1(ts) & S3C2410_ADCDAT1_YPDATA_MASK; /* 读 Y 坐标并累加 */
    ts->count++; /* 采样次数加 1 */
    if (ts->count < NR_SAMPLING) { /* 如果未达到预定的采样次数 */
        set_adc_autoxy(ts);
        start_adc(ts); /* 启动 ADC 进行转换 */
    } else { /* 否则 */

```

```

        mod_timer(&ts->timer, jiffies+HZ/100); /* 启动定时器 */
    }
    return IRQ_HANDLED;
}

static struct input_dev *dev; /* 输入设备 */

/* 初始化函数 */
static int __init ts_init(void)
{
    int err = -ENOMEM;
    struct ts_data *ts;
    /* 给 ts 数据分配空间 */
    ts = kzalloc(sizeof(struct ts_data), GFP_KERNEL);
    if (!ts) goto kzalloc_ts_fail;
    /* 给输入设备分配空间 */
    dev = input_allocate_device();
    if (!dev) goto input_allocate_fail;
    /* 获取并打开时钟源 */
    ts->clk = clk_get(NULL, "adc");
    if (!ts->clk) {
        pr_err("clk_get(ad) ERR!\n");
        err = -ENOENT;
        goto clk_get_fail;
    }
    clk_enable(ts->clk);
    /* 设置 GPIO 管脚的功能 */
    s3c2410_gpio_cfgpin(S3C2410_GPG12, S3C2410_GPG12_XMON);
    s3c2410_gpio_cfgpin(S3C2410_GPG13, S3C2410_GPG13_nXPON);
    s3c2410_gpio_cfgpin(S3C2410_GPG14, S3C2410_GPG14_YMON);
    s3c2410_gpio_cfgpin(S3C2410_GPG15, S3C2410_GPG15_nYPON);
    /* 申请 IO 内存 */
    if (!request_mem_region(S3C24XX_PA_ADC, S3C24XX_SZ_ADC, DEVNAME)) {
        pr_err("ts_init: request_mem_region() ERR!\n");
        err = -EBUSY;
        goto request_mem_fail;
    }
    /* 映射 IO 内存 */
    ts->iobase = ioremap(S3C24XX_PA_ADC, S3C24XX_SZ_ADC);
    if (!ts->iobase) {
        pr_err("ts_init: ioremap() ERR!\n");
        goto ioremap_fail;
    }
    /* 初始化定时器 */
    setup_timer(&ts->timer, ts_timer, (unsigned long)dev);
    /* 初始化数据 */
    clear_data(ts);
    /* 初始化输入设备 */
    dev->name = DEVNAME; /* 名称 */
    /* 设置输入设备的私有数据 */
    input_set_drvdata(dev, ts);
}

```

```

/* 设置输入设备的功能 */
input_set_capability(dev, EV_KEY, BTN_TOUCH);
input_set_capability(dev, EV_ABS, ABS_X);
input_set_capability(dev, EV_ABS, ABS_Y);
/* 设置绝对坐标事件参数 */
input_set_abs_params(dev, ABS_X, X_AXIS_MIN, X_AXIS_MAX, 0, 0);
input_set_abs_params(dev, ABS_Y, Y_AXIS_MIN, Y_AXIS_MAX, 0, 0);
/* 申请中断 */
if (request_irq(IRQ_ADC, adc_handler, IRQF_DISABLED, DEVNAME, dev)) {
    pr_err("request_irq(IRQ_ADC) ERR!\n");
    err = -EBUSY;
    goto request_irq_adc_fail;
}
if (request_irq(IRQ_TC, tc_handler, IRQF_DISABLED, DEVNAME, dev)) {
    pr_err("request_irq(IRQ_TC) ERR!\n");
    err = -EBUSY;
    goto request_irq_tc_fail;
}
/* 注册输入设备 */
if (input_register_device(dev)) {
    pr_err("input_register_device ERR!\n");
    goto input_register_fail;
}
/* 设置 ADC 时钟的预分频值, 以下设置必须放在 IO 内存映射之后 */
set_adc_prescaler(ts, ADC_PRESCALER);
/* 设置转换延时 */
set_adc_delay(ts, ADC_DELAY);
/* 设置 ADC 为等待中断模式 */
set_adc_wait4int(ts);
/* 设置 ADC 为非备用模式 */
set_adc_standby(ts, 0);
/* 关闭 ADC 的读触发功能 */
set_adc_read_start(ts, 0);
return 0;
input_register_fail:
    free_irq(IRQ_TC, dev);
request_irq_tc_fail:
    free_irq(IRQ_ADC, dev);
request_irq_adc_fail:
    iounmap(ts->iobase);
ioremap_fail:
    release_mem_region(S3C24XX_PA_ADC, S3C24XX_SZ_ADC);
request_mem_fail:
    clk_disable(ts->clk);
    clk_put(ts->clk);
clk_get_fail:
    input_free_device(dev);
input_allocate_fail:
    kfree(ts);
kzalloc_ts_fail:
    return err;

```

```
}
module_init(ts_init);

/* 退出函数 */
static void __exit ts_exit(void)
{
    struct ts_data *ts = input_get_drvdata(dev);
    input_unregister_device(dev);
    free_irq(IRQ_TC, dev);
    free_irq(IRQ_ADC, dev);
    del_timer_sync(&ts->timer);
    iounmap(ts->iobase);
    release_mem_region(S3C24XX_PA_ADC, S3C24XX_SZ_ADC);
    clk_disable(ts->clk);
    clk_put(ts->clk);
    kfree(ts);
}
module_exit(ts_exit);
```

16.1.2.5 触摸屏例程说明

为了使上报的坐标稳定，例程中使用了多次采样求平均值的办法。如果采样数为 2 的整数次幂，那么除法操作会被优化为移位操作，提高执行效率。必须注意，只有在除数是常数的情况下编译器才能做这种优化。

例程中，当对数据的采样达到规定的次数时，并没有立刻将坐标上报，而是经过一个定时器周期后，先检测铁笔是否为落下状态，如果为落下状态才将数据上报。这样做虽然使数据的上报有一定的延迟，但实际是合理的。这是因为，在铁笔离开屏的过程中，采样到的坐标会有急剧的变化，甚至有可能采样到铁笔离开屏后的电压值，这个值显然是无效的，将这个值上报实际上是一种对正常坐标的干扰。经过一定时间后判断铁笔状态，再将有效数据上报，这样就可以避免这种情况。

同理，铁笔落下时，经过一个定时器周期后再来判断铁笔是否为落下状态，如果为落下状态才启动 ADC 进行采样，这样可以避免落下过程中产生的抖动对数据造成不利的影响。

在嵌入式 CPU 中，一般会有多个硬件上的时钟源，供不同的外围设备使用。将时钟源关闭可以降低 CPU 的功耗。一个时钟源也可能是多个设备共用，并且某些时钟源可能是从它的父时钟源分频得到的。内核中为此设计了时钟源管理体系，时钟源之间有父子关系，并且支持引用计数。在启动阶段，由机器的初始化代码向内核注册时钟源。源码中的 `clk_get` 和 `clk_put` 函数就是在操作时钟源的引用计数，而 `clk_enable` 和 `clk_disable` 则是在打开和关闭时钟源。注意调用 `clk_disable` 不一定会真正关闭一个时钟源，因为其他设备可能正在使用这个时钟源或它的子时钟源。

本例程在 Linux 2.6.30 内核上编译并测试通过。

16.2 USB 驱动

自从 USB 接口诞生以来，以其支持热插拔的特性，及灵活的可扩展性和适应性，迅速地成为了使用最广泛的计算机外部接口。本节将对 USB 的编程接口进行简单的介绍，并对内核提供的

USB 驱动范例进行分析。

16.2.1 USB 概述

USB 是一种使用差分信号来传输数据的高速串行传输总线，物理层的最高传输速度可达 480Mb/s。USB 通信协议是一种主从式的协议。一个完整的 USB 传输体系包括一个主机、若干个设备及物理连接三个部分。主机是一个提供 USB 接口并且能够对接口进行管理的计算机系统，一个 USB 传输体系内只能有一个主机；设备包括 USB 功能设备和 USB 集线器，前者是有具体功能的设备，后者则能对主机的端口进行扩展；物理连接则是 USB 传输线。

USB 协议使用设备地址来区分各个设备，最多可以支持 127 个可用地址。但由于连接较多设备时一定会用到 USB 集线器，所以实际可以同时连接的功能设备不到 127 个。同时，集线器相互级联的层数被限制在 5 层以内。

USB 协议采用轮询的广播机制传输数据，所有的传输都是由主机发起的，任何时刻整个 USB 传输体系内仅允许一个数据包的传输。

USB 设备与主机通信的最小逻辑通道是管道，管道在设备上的终点称为端点。一个设备可以同时有几个端点与主机进行通信，它们由端点地址来区分。端点地址包含了端点的编号和端点的传输方向。端点编号的取值是 0~15 范围内的整数。地址为 0 的端点是一个特殊的端点，它可以进行双向的传输，这个端点是任何设备都必须有的，因为设备的控制信息要通过它来传输。其他的端点则只能进行单向的传输。

因为除 0 号端点之外的所有端点只能进行单向的传输，所以要实现一个具体的功能往往需要把多个端点组合起来使用，这被称为是设备的接口。接口是设备具体功能的最小单位。接口还可以有多个可选设置，不同的设置下可以有不同的传输参数，但同一时刻只有一个设置是生效的。

复杂的设备可能同时拥有多个接口，并且还能支持不同的配置。在不同的配置下，一个设备拥有的接口数量和类型有可能是不同的，也就是说，不同的配置导致设备有不同的功能。设备、配置、接口与端点的关系可由图 16.4 表示，并简单地总结如下。

- ◆ 一个设备可以有好几套不同的配置。
- ◆ 一个配置由若干个接口组成。
- ◆ 一个接口由若干个端点组成。
- ◆ 一个接口可以有多个不同设置。

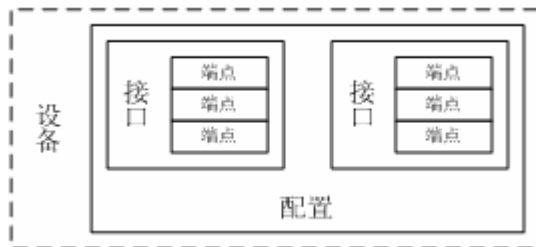


图 16.4 设备、配置、接口与端点的关系

管道的传输方式可分为四种类型，对应的端点也分为这四种类型，分别描述如下。

一、控制（control）型

控制型传输是一种可靠的双向传输方式，主机通过它获取设备的信息，并对设备进行配置。任何设备都必须有一个地址为 0 的控制型端点。

二、中断（interrupt）型

中断型传输是一种可靠的单向传输方式。主机对中断端点进行定期的轮询，若有数据可以传输则进行传输，以此来模拟设备的“中断”。适用于对传输延迟有较严格的要求、数据量较小的传输。

三、块（bulk）型

块型传输是一种可靠的单向传输方式，但对传输延迟没有控制。它可以尽可能地利用总线的空闲带宽来进行传输。适用于对传输延迟的要求比较宽松、数据量较大的传输。

四、等时（isochronous）型

等时型传输是一种不可靠但是传输延迟最小的单向传输方式，它可以定时定量地进行数据的传输，但所传输的数据包是有可能丢失的。等时传输时主机也要对端点定期进行轮询。适用于对可靠性要求不高的实时数据传输，如音频数据。

当一个 USB 设备连接到主机时，主机将通过默认的控制型管道对其进行查询，获取其配置、接口、端点等的描述，为其分配地址并进行配置操作，这个过程称为 USB 设备的枚举。枚举成功之后设备才能正常使用。

在枚举过程中，设备的很多信息被读取出来，其中包括两个很重要的数据：制造商识别码（VID）和产品识别码（PID）。制造商识别码是一个 16 位的数据，用来区别各个不同的 USB 设备制造商，它的取值须由制造商向 USB 执行论坛（USB Implementers Forum）公司申请；产品识别码也是一个 16 位的数据，由产品的制造商自行决定。

16.2.2 USB 驱动模型

Linux 上的 USB 驱动模型如图 16.5 所示。

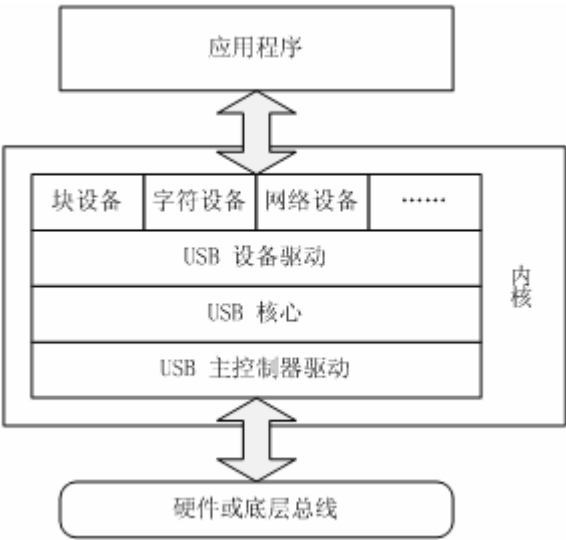


图 16.5 USB 驱动模型

在硬件上, USB 总线由 USB 主控制器这个设备支持, 因此, 软件上必须有相应的主控制器驱动。USB 核心指的是内核中关于 USB 协议的核心代码, 它对上提供透明的直接面向 USB 功能设备的管理接口。USB 设备驱动则是针对某种具体的 USB 功能设备的驱动, 它对设备进行探测并将其进一步抽象为直接面向应用程序的设备。

由于 USB 设备的接口代表了一种具体的功能, 各个接口之间通常没有相互联系, 因此 USB 核心提供了面向接口的驱动编程方式, 由 USB 核心提供一个通用的 USB 设备驱动, 这个驱动将把 USB 设备内的各个接口抽象为自己的子设备进行注册。这样, 驱动编程者只需要提供对应各个接口的 USB 接口驱动即可。一般所说的 USB 驱动实际上指的就是 USB 接口驱动。

Linux 是这样来管理 USB 设备和驱动的: 在内核启动阶段, 由 USB 核心向系统注册 USB 总线。当新的 USB 设备接入时, 由 USB 核心进行枚举过程并最终向 USB 总线注册设备。USB 主控制器也被作为 USB 设备向 USB 总线注册。

在 USB 协议中定义了一些标准的设备类别, 如输入设备类(键盘、鼠标等)、存储设备类(U 盘、移动硬盘等)、音频设备类等, 对于这些标准的设备, 内核中都提供了相应的设备驱动, 因而不需要再另外写驱动。在大多数情况下, 只有那些特制的 USB 设备才需要为其开发驱动。

16.2.3 USB 驱动编程接口

与 USB 驱动编程有关的接口的定义和声明都在以下头文件中:

```
#include <linux/usb.h>
```

16.2.3.1 USB 驱动数据类型

USB 驱动由以下数据结构表示:

```
struct usb_driver {
    const char *name; /* 名称 */
    /* 探测操作 */
    int (*probe)(struct usb_interface *intf, const struct usb_device_id *id);
    void (*disconnect)(struct usb_interface *intf); /* 断开操作 */
    /* 挂起操作 */
    int (*suspend)(struct usb_interface *intf, pm_message_t message);
    int (*resume)(struct usb_interface *intf); /* 恢复操作 */
    int (*reset_resume)(struct usb_interface *intf); /* 挂起状态下的复位操作 */
    int (*pre_reset)(struct usb_interface *intf); /* 复位前操作 */
    int (*post_reset)(struct usb_interface *intf); /* 复位后操作 */
    const struct usb_device_id *id_table; /* 匹配列表 */
    struct usbdrv_wrap drvwrap; /* 内核内部使用 */
    unsigned int supports_autosuspend:1; /* 是否支持自动挂起 */
    /* 这里只写出了部分成员 */
};
```

其中 struct usbdrv_wrap 类型的定义如下:

```
struct usbdrv_wrap {
    struct device_driver driver; /* 内嵌的驱动类型 */
};
```

```
int for_devices; /* 1 表示是 USB 设备驱动, 0 表示是 USB 驱动 */
};
```

可见 USB 驱动最终也包含一个内嵌的驱动类型数据, 作为 USB 驱动的代表向 USB 总线注册。

USB 驱动类型中, `name`, `probe`, `disconnect` 和 `id_table` 四个成员是必须有初值的, 其他则为可选。`name` 表示驱动的名称, 一般就用模块的名称以使它在全局唯一。`probe` 操作在 USB 接口与驱动匹配后被调用, `disconnect` 操作在设备注销 (被拔出) 及驱动注销时调用。

16.2.3.2 USB 驱动的匹配列表

USB 驱动的 `id_table` 表示一个匹配列表, 其中的每个条目都表示一个匹配项。当进行设备与驱动的匹配时, 其中的匹配项将按次序逐个进行检验, 设备的信息只要符合其中一个匹配项就算匹配成功。编程时一般并不关心匹配项的具体内容, 而用内核提供的以下几个宏来构造匹配项:

```
USB_DEVICE(vid, pid)
USB_DEVICE_VER(vid, pid, lo, hi)
USB_DEVICE_INTERFACE_PROTOCOL(vid, pid, intf_proto)
USB_DEVICE_INFO(dev_class, dev_subclass, dev_proto)
USB_INTERFACE_INFO(intf_class, intf_subclass, intf_proto)
USB_DEVICE_AND_INTERFACE_INFO(vid, pid, intf_class, intf_subclass,
                               intf_proto)
```

其中各个参数的含义解释如下。

- ◆ `vid`: 制造商识别码。
- ◆ `pid`: 产品识别码。
- ◆ `lo`: 版本号范围的最小值。
- ◆ `hi`: 版本号范围的最大值。
- ◆ `intf_class`: 接口类别。
- ◆ `intf_subclass`: 接口子类别。
- ◆ `intf_proto`: 接口协议。
- ◆ `dev_class`: 设备类别。
- ◆ `dev_subclass`: 设备子类别。
- ◆ `dev_proto`: 设备协议。

宏里面包含哪些参数就表示通过哪些参数来匹配, 例如:

```
static struct usb_device_id skel_table [] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    { } /* 列表结束 */
};
```

这个匹配列表中定义了一个通过 `PID` 和 `VID` 进行匹配的条目。数组的最后一个元素必须为空以表示列表结束。通常对于匹配列表还要做一个类似如下的声明:

```
MODULE_DEVICE_TABLE(usb, skel_table);
```

这个宏的作用是在编译内核的过程中将匹配列表的内容导出到文件 `modules.usbmap` 中，管理热插拔的应用程序可以根据这个文件的内容决定加载哪个驱动模块。

16.2.3.3 USB 驱动的注册与注销

驱动初始化好之后就可以向 USB 总线注册了，所用的接口函数原型如下：

```
int usb_register(struct usb_driver *driver);
```

其中参数 `driver` 指向要注册的驱动，返回值为 0 表示注册成功，为负数则表示一个错误码。相反的操作是从 USB 总线注销驱动，所用的接口函数原型如下：

```
void usb_deregister(struct usb_driver *driver);
```

其中参数 `driver` 指向要注销的驱动。

16.2.4 USB 接口与端点

16.2.4.1 USB 接口数据类型与操作

USB 驱动的探测操作的第一个传入参数是表示接口的 `struct usb_interface` 类型，第二个参数则指向成功的匹配项。接口对应的数据类型的定义如下：

```
struct usb_interface {
    struct usb_host_interface *altsetting; /* 可选设置列表 */
    struct usb_host_interface *cur_altsetting; /* 当前设置 */
    unsigned num_altsetting; /* 可选设置个数 */
    struct device dev; /* 接口的内嵌设备类型 */
    struct device *usb_dev; /* 指向与接口绑定的 USB 类别设备 */
    /* 这里只写出了部分成员 */
};
```

可见接口数据类型中也内嵌了一个设备对象，说明接口在设备模型中也是被抽象为一个单独设备的。内核提供了一个宏，可以由指向设备对象的指针得到指向接口的指针：

```
struct usb_interface *to_usb_interface(struct device *dev);
```

内核还提供了一个宏，可以由接口得到接口所属的 USB 设备：

```
struct usb_device *interface_to_usbdev(struct usb_interface *intf);
```

注意它的返回值指向 `struct usb_device` 类型的数据，这个类型代表 USB 设备。从设备模型的角度看，它是这个接口的父设备。

内核还提供了两个函数以便驱动将自己的私有数据与接口绑定，它们的定义如下：

```
static inline void *usb_get_intfdata(struct usb_interface *intf)
{
    return dev_get_drvdata(&intf->dev);
}

static inline void usb_set_intfdata(struct usb_interface *intf, void *data)
```

```
{
    dev_set_drvdata(&intf->dev, data);
}
```

可见实际上它们利用了设备对象中的驱动私有数据。

接口也支持引用计数的相关操作，为此内核中定义了以下两个函数：

```
struct usb_interface *usb_get_intf(struct usb_interface *intf)
{
    if (intf) get_device(&intf->dev);
    return intf;
}

void usb_put_intf(struct usb_interface *intf)
{
    if (intf) put_device(&intf->dev);
}
```

这里仍然在利用接口内嵌的设备对象。

`struct usb_host_interface` 是描述接口具体设置的数据类型，定义如下：

```
struct usb_host_interface {
    struct usb_interface_descriptor desc; /* 接口描述符 */
    struct usb_host_endpoint *endpoint; /* 端点列表（个数在接口描述符内） */
    /* 这里只写出了部分成员 */
};
```

接口描述符对应的数据类型定义如下：

```
struct usb_interface_descriptor {
    __u8 bLength; /* 描述符长度 */
    __u8 bDescriptorType; /* 描述符类型 */
    __u8 bInterfaceNumber; /* 接口编号 */
    __u8 bAlternateSetting; /* 可选设置编号 */
    __u8 bNumEndpoints; /* 端点个数 */
    __u8 bInterfaceClass; /* 接口类别 */
    __u8 bInterfaceSubClass; /* 接口子类别 */
    __u8 bInterfaceProtocol; /* 接口协议 */
    __u8 iInterface; /* 描述接口的字符串索引 */
} __attribute__((packed));
```

注意这里的变量命名方式不再遵循内核的全小写命名惯例，因为这些数据域都是在 USB 协议中定义的。由于这些数据是从 USB 设备读取的，因此必须定义为按字节对齐的紧凑格式。

16.2.4.2 USB 端点数据类型与操作

`struct usb_host_endpoint` 是描述端点的数据类型，其定义如下：

```
struct usb_host_endpoint {
    struct usb_endpoint_descriptor desc; /* 端点描述符 */
    /* 这里只写出了部分成员 */
}
```

```
};
```

它的主要成员是一个端点描述符，这又是一个从硬件读取的数据类型，定义如下：

```
struct usb_endpoint_descriptor {
    __u8  bLength; /* 描述符长度 */
    __u8  bDescriptorType; /* 描述符类型 */
    __u8  bEndpointAddress; /* 端点地址 */
    __u8  bmAttributes; /* 端点属性 */
    __le16 wMaxPacketSize; /* 允许的最大包长 */
    __u8  bInterval; /* 查询间隔（仅对中断型与等时型端点） */
    __u8  bRefresh; /* 刷新周期（仅对等时型端点） */
    __u8  bSynchAddress; /* 同步地址（仅对等时型端点） */
} __attribute__((packed));
```

实际使用时，这个数据类型的最后两个成员可能是无效的，因此要得到端点描述符的实际大小，不能用 `sizeof(struct usb_endpoint_descriptor)`，而要用下面的宏：

```
#define USB_DT_ENDPOINT_SIZE      7 /* 非等时型端点 */
#define USB_DT_ENDPOINT_AUDIO_SIZE 9 /* 等时型端点 */
```

大多数情况下并不需要关心端点描述符的具体内容，因为内核提供了大量的函数对端点描述符进行处理，例如使用下面这个函数可以得到端点的编号：

```
int usb_endpoint_num(const struct usb_endpoint_descriptor *epd);
```

其返回值是 0~15 范围内的整数。端点的编号和端点的地址并不等同。实际上，地址的低 4 位表示编号，而最高一位则表示端点的传输方向。

下面这个函数可以得到端点的类型：

```
int usb_endpoint_type(const struct usb_endpoint_descriptor *epd);
```

其返回值是以下四个值之一。

- ◆ USB_ENDPOINT_XFER_CONTROL：控制型。
- ◆ USB_ENDPOINT_XFER_ISOC：等时型。
- ◆ USB_ENDPOINT_XFER_BULK：块型。
- ◆ USB_ENDPOINT_XFER_INT：中断型。

下面这些函数可以判断端点的传输方向是否是输入或输出：

```
/* 判断是否是输入 */
int usb_endpoint_dir_in(const struct usb_endpoint_descriptor *epd);
/* 判断是否是输出 */
int usb_endpoint_dir_out(const struct usb_endpoint_descriptor *epd);
```

下面这些函数可以判断端点的类型是否是指定类型：

```
/* 判断是否是控制型 */
int usb_endpoint_xfer_control(const struct usb_endpoint_descriptor *epd);
```



```
/* 判断是否是中断型 */
int usb_endpoint_xfer_int(const struct usb_endpoint_descriptor *epd);
/* 判断是否是块型 */
int usb_endpoint_xfer_bulk(const struct usb_endpoint_descriptor *epd);
/* 判断是否是等时型 */
int usb_endpoint_xfer_isoc(const struct usb_endpoint_descriptor *epd);
```

下面这些函数可以同时判断端点的类型和传输方向是否是指定的类型和方向：

```
/* 判断是否是中断输入型 */
int usb_endpoint_is_int_in(const struct usb_endpoint_descriptor *epd);
/* 判断是否是中断输出型 */
int usb_endpoint_is_int_out(const struct usb_endpoint_descriptor *epd);
/* 判断是否是块输入型 */
int usb_endpoint_is_bulk_in(const struct usb_endpoint_descriptor *epd);
/* 判断是否是块输出型 */
int usb_endpoint_is_bulk_out(const struct usb_endpoint_descriptor *epd);
/* 判断是否是等时输入型 */
int usb_endpoint_is_isoc_in(const struct usb_endpoint_descriptor *epd);
/* 判断是否是等时输出型 */
int usb_endpoint_is_isoc_out(const struct usb_endpoint_descriptor *epd);
```

16.2.5 USB 类别

驱动探测到合适的 USB 接口以后，就要为这个接口注册合适的类别设备。如果是存储设备，则注册一个块设备；如果是键盘、鼠标，则注册输入设备……如果只是一个简单的字符设备，则可以注册一个 USB 类别设备。

16.2.5.1 USB 类别设备

USB 类别非常类似于 misc 类别，它占据主设备号 USB_CHAR_MAJOR，定义如下：

```
#define USB_CHAR_MAJOR    180
```

表示 USB 类别设备的数据类型定义如下：

```
struct usb_class_driver {
    char *name; /* 设备名称 */
    const struct file_operations *fops; /* 文件操作 */
    int minor_base; /* 次设备号起始值 */
};
```

这里只需要提供设备名称和文件操作即可。设备名称中可以使用一个 %d 格式字符串，注册后 %d 会被替换为设备的编号，如 skel%d 对应的设备名称将是 skel0, skel1 等。minor_base 只是一个建议值，注册类别设备时内核会以 minor_base 作为起始值，向上搜索空闲的次设备号来使用，所以这个值设为 0 就等价于由内核决定次设备号，如果编译时配置了 USB 动态次设备号选项，即：

```
CONFIG_USB_DYNAMIC_MINORS=y
```



则 `minor_base` 的取值完全无用。

表示 USB 类别设备的数据类型也往往被称为 USB 类别驱动。实际上，字符设备的设备数据和驱动是一体的，无法分离。

16.2.5.2 USB 类别设备的注册与注销

注册 USB 类别设备的接口函数原型如下：

```
int usb_register_dev(struct usb_interface *intf,
                    struct usb_class_driver *class_driver);
```

其各个参数及返回值的含义解释如下。

- ◆ `intf`：与类别设备对应的接口。
- ◆ `class_driver`：要注册的 USB 类别驱动。
- ◆ 返回值：0 表示注册成功，负数表示错误码。

注册之后，参数 `intf` 指向的接口的 `usb_dev` 成员就会指向新创建的设备对象。

与注册相反的操作是注销。注销 USB 类别设备的接口函数原型如下：

```
void usb_deregister_dev(struct usb_interface *intf,
                      struct usb_class_driver *class_driver);
```

其中参数 `intf` 指向类别设备所绑定的接口，参数 `class_driver` 则指向要注销的类别驱动。

通过类别设备的次设备号还可以反查对应的接口：

```
struct usb_interface *usb_find_interface(struct usb_driver *drv, int minor);
```

其各个参数和返回值的含义解释如下。

- ◆ `drv`：指向注册 USB 类别设备的 USB 驱动。
- ◆ `minor`：次设备号。
- ◆ 返回值：指向对应的 USB 接口，NULL 表示出错或找不到。

要注意这个函数只对注册了的 USB 类别设备有效，注册为其他类别的设备则不能用这个函数来查找对应的接口。

16.2.6 URB

不管 USB 设备最终注册为何种类别设备，都不可避免地要经过 USB 核心与硬件设备进行数据交换。USB 核心提供了基本的输入输出接口 URB (USB Request Block, USB 请求块)。URB 是一个庞大的结构体，包含了用于 USB 输入输出的所有必要信息。USB 驱动通过向 USB 核心提交 URB 的方式来启动输入输出过程。

16.2.6.1 URB 数据类型

URB 数据类型的定义如下：



```

struct urb {
    struct kref kref; /* 引用计数 */
    atomic_t reject; /* 废弃标志 */
    struct list_head anchor_list; /* 链表节点成员，用来加入 USB 锚的链表 */
    struct usb_anchor *anchor; /* 指向锚定它的 USB 锚 */
    struct usb_device *dev; /* 指向所属的 USB 设备 */
    struct usb_host_endpoint *ep; /* 指向对应的端点 */
    unsigned int pipe; /* 管道号 */
    int status; /* URB 的状态 */
    unsigned int transfer_flags; /* 传输标志 */
    void *transfer_buffer; /* 传输缓冲区 */
    dma_addr_t transfer_dma; /* 传输缓冲区的 DMA 地址 */
    u32 transfer_buffer_length; /* 传输缓冲区长度 */
    u32 actual_length; /* 实际传输的字节数 */
    unsigned char *setup_packet; /* 8 字节的请求包缓冲区（仅对控制型端点） */
    dma_addr_t setup_dma; /* 请求包 DMA 地址 */
    int start_frame; /* 等时传输包的起始帧 */
    int number_of_packets; /* 等时传输包的个数 */
    int interval; /* 查询间隔（仅对中断型和等时型） */
    int error_count; /* 等时传输的错误个数 */
    void *context; /* 完成函数上下文 */
    usb_complete_t complete; /* 完成函数 */
    /* 这里只写出了部分成员 */
};

```

这些成员并不是全部由驱动来初始化的，有些是由 USB 核心使用的。

16.2.6.2 URB 的分配与释放

用下面的函数可以动态地分配一个 URB：

```
struct urb *usb_alloc_urb(int iso_packets, gfp_t mem_flags);
```

其各个参数及返回值的含义如下。

- ◆ iso_packets：需要的等时包的个数（仅对等时型传输有用）。
- ◆ mem_flags：分配内存的标志，与 kmalloc 函数的第二个参数含义相同。
- ◆ 返回值：指向分配到的 URB，NULL 表示分配失败。

动态分配到的 URB 最后需要用以下函数释放：

```
void usb_free_urb(struct urb *urb);
```

其中参数 urb 指向要释放的 URB。注意这个函数实质上仅仅是减少对 URB 的引用计数。当 URB 提交时，USB 核心会增加对它的引用计数，因此，URB 正确提交后就可以被释放。最终 URB 所占内存的释放在 URB 被完成以后才会执行。为了避免歧义，可使用下面这个函数：

```
void usb_put_urb(struct urb *urb);
```

这两个释放函数是完全相同的。



16.2.6.3 URB 的初始化

在提交 URB 之前，驱动需要将其关键成员初始化，这可以通过直接赋值来完成，也可以通过内核定义的辅助函数来完成。

初始化控制型传输的 URB 可用以下函数：

```
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev,
    unsigned int pipe, unsigned char *setup_packet,
    void *transfer_buffer, int buffer_length,
    usb_complete_t complete_fn, void *context);
```

其各个参数的含义解释如下。

- ◆ urb：指向要初始化的 URB。
- ◆ dev：指向所属的 USB 设备。
- ◆ pipe：传输所用的管道号。
- ◆ setup_packet：指向 8 字节的请求包。
- ◆ transfer_buffer：指向传输缓冲区。
- ◆ buffer_length：传输缓冲区的长度。
- ◆ complete_fn：完成函数，在 URB 传输完成时被调用。
- ◆ context：URB 的上下文参数。

初始化块型传输的 URB 可用以下函数：

```
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev,
    unsigned int pipe,
    void *transfer_buffer, int buffer_length,
    usb_complete_t complete_fn, void *context);
```

它比 `usb_fill_control_urb` 函数少一个参数 `setup_packet`，其余各个参数的含义与之相同。

初始化中断型传输的 URB 可用以下函数：

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev,
    unsigned int pipe,
    void *transfer_buffer, int buffer_length,
    usb_complete_t complete_fn, void *context, int interval);
```

与 `usb_fill_bulk_urb` 函数相比，它多了一个参数 `interval`，这个参数表示轮询的周期。对于低速和全速设备来说，单位是帧（1 毫秒）；对于高速设备来说，单位是微帧（1/8 毫秒）。

16.2.6.4 管道号

初始化 URB 时，一件重要的事情就是指明传输所用的管道号。管道号是 USB 核心用来标识传输通道的唯一编号，它与端点的地址有关，可用内核提供的以下函数进行构造：

```
/* 构造控制输出型管道号 */
int usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint);
/* 构造控制输入型管道号 */
```



```
int usb_rcvctrlpipe(struct usb_device *dev, unsigned int endpoint);
/* 构造等时输出型管道号 */
int usb_sndisocpipe(struct usb_device *dev, unsigned int endpoint);
/* 构造等时输入型管道号 */
int usb_rcvisocpipe(struct usb_device *dev, unsigned int endpoint);
/* 构造块输出型管道号 */
int usb_sndbulkpipe(struct usb_device *dev, unsigned int endpoint);
/* 构造块输入型管道号 */
int usb_rcvbulkpipe(struct usb_device *dev, unsigned int endpoint);
/* 构造中断输出型管道号 */
int usb_sndintpipe(struct usb_device *dev, unsigned int endpoint);
/* 构造中断输入型管道号 */
int usb_rcvintpipe(struct usb_device *dev, unsigned int endpoint);
```

其中参数 `dev` 指向进行传输的 USB 设备，参数 `endpoint` 则是端点的地址。

16.2.6.5 传输缓冲区

USB 设备的传输通常是 DMA 的方式，因此必须将传输缓冲区分配在 DMA 控制器能够识别的内存区域里，这时用 `kmalloc` 就不合适了。内核提供了专门为 USB 设备分配传输缓冲区的函数，原型如下：

```
void *usb_buffer_alloc(struct usb_device *dev, size_t size,
                      gfp_t mem_flags, dma_addr_t *dma);
```

其各个参数及返回值的含义解释如下。

- ◆ `dev`：指向所属的 USB 设备。
- ◆ `size`：缓冲区的大小。
- ◆ `mem_flags`：分配内存的标志，与 `kmalloc` 函数的第二个参数含义相同。
- ◆ `dma`：返回缓冲区对应的 DMA 地址。
- ◆ 返回值：指向分配到的缓冲区，`NULL` 表示失败。

由于通过这个函数分配缓冲区的同时已经得到了 DMA 地址，所以要对 URB 的传输标志进行如下的设置：

```
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

这个标志的作用是通知 USB 主控制器驱动：不需要再为传输缓冲区进行 DMA 的映射，而是直接使用 URB 中提供的 DMA 地址，即 `transfer_dma` 成员的值。对于 8 字节的请求包缓冲区则是另外一个标志：

```
urb->transfer_flags |= URB_NO_SETUP_DMA_MAP;
```

这个标志的作用是通知 USB 主控制器驱动：不需要再为请求包缓冲区进行 DMA 的映射，而是直接使用 URB 中提供的 DMA 地址，即 `setup_dma` 成员的值。

当分配到的缓冲区不再使用时，需要将其释放，使用下面的函数：



```
void usb_buffer_free(struct usb_device *dev, size_t size, void *addr,
                    dma_addr_t dma);
```

其各个参数的含义解释如下。

- ◆ dev: 指向所属的 USB 设备。
- ◆ size: 缓冲区的大小。
- ◆ addr: 指向缓冲区。
- ◆ dma: 缓冲区对应的 DMA 地址。

16.2.6.6 URB 的完成函数

使用 URB 进行传输是一种异步的传输，驱动在提交 URB 以后无须等待传输过程的真正结束。当 USB 核心完成传输过程或者发生错误以后，以调用完成函数的方式来通知驱动。完成函数是由驱动提供的，其指针类型定义如下：

```
typedef void (*usb_complete_t)(struct urb *);
```

这个函数只有一个参数，当 USB 核心回调这个函数时，将被完成的 URB 的指针传入。驱动可以根据 URB 中的信息进行分析，常用的有以下几个变量。

- ◆ urb->status: URB 的状态，0 表示传输成功，负数为错误码。
- ◆ urb->actual_length: 实际传输的字节数。
- ◆ urb->context: 这个值在提交 URB 前由驱动设置，完成时原封不动地传进来。

完成函数的调用与 URB 的提交是一一对应的关系。一个 URB 一旦被提交，即使传输失败或被撤销，也要调用一次完成函数。



完成函数运行在中断上下文中，因此不能使用任何引起进程调度的操作。

16.2.6.7 URB 的提交与撤销

提交 URB 就意味着通知 USB 核心启动传输过程，其接口函数原型如下：

```
int usb_submit_urb(struct urb *urb, gfp_t mem_flags);
```

其各个参数及返回值的含义如下。

- ◆ urb: 指向被提交的 URB。
- ◆ mem_flags: 申请内存的标志，与 kmalloc 函数的第二个参数含义相同。
- ◆ 返回值: 0 表示成功，负数表示错误码。

URB 提交以后就由 USB 核心管理，驱动不能再随意修改它，直到完成函数被调用。

在完成函数中可以提交下一个要传输的 URB，这也是一种常用的编程模式，这时要注意参数 mem_flags 必须用 GFP_ATOMIC。

提交的 URB 还可以被撤销，所用的接口函数原型如下：

```
int usb_unlink_urb(struct urb *urb);
```

其参数及返回值的含义如下。

- ◆ **urb**: 指向被撤销的 URB。
- ◆ **返回值**: **-EINPROGRESS** 表示撤销成功, 其他值表示撤销失败。

注意这里的返回值并不是 0 表示成功。当这个函数返回成功时, 被撤销的 URB 会尽快完成, 完成时 URB 的状态是 **-ECONNRESET**。如果函数返回失败, 则说明无法撤销。

如果要将 URB 撤销, 并且确保这个 URB 不在使用状态, 则必须用下面这个函数:

```
void usb_kill_urb(struct urb *urb);
```

这个函数会发起对 URB 的撤销操作, 并且等待其完成。实际上, 在这个函数返回之前, 完成函数就已经被调用, 传入的 URB 的状态是 **-ENOENT**。这里的等待是由等待队列实现的, 所以这个函数不能用在原子上上下文中。

这个函数还有一个作用就是在等待 URB 完成期间设置它的废弃标志 **reject**。废弃标志不为 0 的 URB 如果被提交会立刻返回失败。也就是说, **usb_kill_urb** 函数可以防止在完成函数中再次提交这个 URB。当这个函数返回后, URB 重新成为可用的。

撤销 URB 还可以用另外一个函数, 它与 **usb_kill_urb** 函数的作用几乎完全相同, 只不过在返回前并没有将废弃标志还原, 因此在它返回后, URB 仍然不可用。其原型如下:

```
void usb_poison_urb(struct urb *urb);
```

为了使 URB 重新成为可用的, 需要使用下面的接口函数:

```
void usb_unpoison_urb(struct urb *urb);
```

16.2.7 同步传输接口

内核中还提供了 USB 的同步传输接口, 它们的特点是函数不立刻返回, 而是等到传输过程完成或者超时后才返回。实际上, 它们内部都是用 URB 的方式实现的。

同步传输控制信息的接口如下:

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe,
    __u8 request, __u8 requesttype, __u16 value, __u16 index,
    void *data, __u16 size, int timeout);
```

其各个参数及返回值的含义解释如下。

- ◆ **dev**: 指向 USB 设备。
- ◆ **pipe**: 传输所用的管道号。
- ◆ **request**: 请求内容。
- ◆ **requesttype**: 请求类型。
- ◆ **value**: 值。
- ◆ **index**: 索引。

- ◆ data: 指向数据缓冲区。
- ◆ size: 数据缓冲区的大小。
- ◆ timeout: 超时限制, 单位是毫秒。
- ◆ 返回值: 传输成功的字节数, 负数表示错误码。

requesttype, request, value, index 和 size 这 5 个数据组合起来就是 8 字节的请求包。同步传输中断信息的接口如下:

```
int usb_interrupt_msg(struct usb_device *usb_dev, unsigned int pipe,
    void *data, int len, int *actual_length, int timeout);
```

其各个参数及返回值的含义解释如下。

- ◆ usb_dev: 指向 USB 设备。
- ◆ pipe: 传输所用的管道号。
- ◆ data: 指向数据缓冲区。
- ◆ len: 数据缓冲区的大小。
- ◆ actual_length: 用于返回实际传输的字节数。
- ◆ timeout: 超时限制, 单位是毫秒。
- ◆ 返回值: 0 表示成功, 负数表示错误码。

同步传输块信息的接口如下:

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
    void *data, int len, int *actual_length, int timeout);
```

这个函数与 usb_interrupt_msg 函数的原型完全相同。实际上, 因为端点类型完全可以由参数 usb_dev 和 pipe 决定, 所以在这两个函数内完全可以决定传输类型, 从而采用合适的初始化方式。

对于控制类型的信息, 内核还将它包装成了一些具体的函数, 比如读取设备的各种描述符可用下面这个函数:

```
int usb_get_descriptor(struct usb_device *dev, unsigned char desc_type,
    unsigned char desc_index, void *buf, int size);
```

读取设备的状态可用下面这个函数:

```
int usb_get_status(struct usb_device *dev, int type, int target, void *data);
```

读取设备字符串可用下面这个函数:

```
int usb_string(struct usb_device *dev, int index, char *buf, size_t size);
```

16.2.8 USB 锚

当 USB 设备被拔出或者 USB 驱动被卸载时会调用驱动的 disconnect 操作, 这时需要将所有已经提交但尚未完成的 URB 撤销, 以便释放所占用的资源。这就需要驱动对提交的 URB 进行

管理，比如将所有提交的 URB 放在一个队列里，每完成一个，就将它从队列里去掉。鉴于这种管理有普遍意义，内核提供了专门的数据类型和接口函数来进行操作，这就是 USB 锚。

代表 USB 锚的数据类型定义如下：

```
struct usb_anchor {
    struct list_head urb_list; /* 链表头，用于链住所有被锚定的 URB */
    wait_queue_head_t wait; /* 等待队列，用于等待被锚定的 URB 离开队列 */
    spinlock_t lock; /* 对成员的访问进行同步 */
    unsigned int poisoned:1; /* 是否将锚定的 URB 废弃 */
};
```

USB 锚的各个成员必须先初始化才能使用，可以用下面这个函数：

```
void init_usb_anchor(struct usb_anchor *anchor);
```

其中参数 `anchor` 指向被初始化的 USB 锚。

USB 锚可以将一个 URB 记录在链表中，称为对这个 URB 的锚定，所用的接口函数原型如下：

```
void usb_anchor_urb(struct urb *urb, struct usb_anchor *anchor);
```

其中参数 `urb` 指向被锚定的 URB，参数 `anchor` 则指向 USB 锚。

相反的操作是解除锚定，这将使一个 URB 从 USB 锚的记录中消除，所用的接口函数原型如下：

```
void usb_unanchor_urb(struct urb *urb);
```

其中参数 `urb` 指向被解除锚定的 URB。因为在 URB 数据类型里有一个成员（`anchor`）保存了锚定它的 USB 锚的指针，所以这里不需要再指明是哪一个 USB 锚。USB 核心在调用完成函数之前会自动将 URB 解除锚定，因此在完成函数里不用再对 URB 调用这个函数。

USB 锚的最大用途在于对所有锚定的 URB 一起进行相同的操作，比如将所有被锚定的 URB 全部撤销可用下面这个函数：

```
void usb_unlink_anchored_urbs(struct usb_anchor *anchor);
```

如果要将所有被锚定的 URB 全部撤销，并且等待它们的完成则可以用下面这个函数：

```
void usb_kill_anchored_urbs(struct usb_anchor *anchor);
```

也可以用另外一个函数：

```
void usb_poison_anchored_urbs(struct usb_anchor *anchor);
```

这个函数可以将所有被锚定的 URB 全部撤销，等待它们的完成并且将它们标记为废弃状态。它还会设置 USB 锚的废弃标志，这样以后所有新被锚定的 URB 也立刻成为废弃状态。如果要将它们全部恢复为可用状态，可以用下面这个函数：

```
void usb_unpoison_anchored_urbs(struct usb_anchor *anchor);
```

这个函数可以将所有被锚定的 URB 从废弃状态变为可用状态，并且消除 USB 锚的废弃标志。

下面这个函数可以等待所有被锚定的 USB 的完成（离开 USB 锚）：

```
int usb_wait_anchor_empty_timeout(struct usb_anchor *anchor,
    unsigned int timeout);
```

其中参数 `timeout` 是超时限制，单位是毫秒。返回值的含义与 `wait_event_timeout` 函数相同。

下面这个函数可以判断 USB 锚中是否包含锚定的 URB：

```
int usb_anchor_empty(struct usb_anchor *anchor);
```

如果这个函数返回非 0 值，说明参数 `anchor` 指向的 USB 锚是空的。

16.2.9 USB 驱动范例分析

内核中提供了一个 USB 驱动的原型。这个驱动是可以编译并使用的，它能够管理有一个块输出端点和一个块输入端点的 USB 接口，并将其注册为一个字符设备。由于 USB 驱动比较复杂，如果从内核提供的范例出发，可以大大减轻开发所需的工作量。以下是内核中 USB 驱动范例的源码，只是由笔者重新加了注释。

```
/* 文件名: drivers/usb/usb-skeleton.c */
/* 说明: Linux 2.6.30 内核 USB 驱动范例 */

#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/kref.h>
#include <asm/uaccess.h>
#include <linux/usb.h>
#include <linux/mutex.h>

#define USB_SKEL_VENDOR_ID 0xffff0 /* 制造商识别码 */
#define USB_SKEL_PRODUCT_ID 0xffff0 /* 产品识别码 */

/* 匹配列表, 通过 VID, PID 匹配 */
static struct usb_device_id skel_table [] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    { } /* 列表结束 */
};

MODULE_DEVICE_TABLE(usb, skel_table); /* 导出匹配列表 */

#define USB_SKEL_MINOR_BASE 192 /* 次设备号的起始值 */

#define MAX_TRANSFER (PAGE_SIZE - 512) /* 包的最大长度 */
/* 这个值的选取既要考虑不能超过一页内存的大小, 以免降低内存的分配效率,
 * 又要考虑到是一个传输包大小的整数倍以提高内存的利用率 */

#define WRITES_IN_FLIGHT 8 /* 传输中的包的个数限制 */

/* 设备数据 */
```

```

struct usb_skel {
    struct usb_device      *udev; /* 指向所属的 USB 设备 */
    struct usb_interface  *interface; /* 指向接口 */
    struct semaphore       limit_sem; /* 用于限制传输中的包的个数 */
    struct usb_anchor       submitted; /* USB 锚 */
    unsigned char          *bulk_in_buffer; /* 输入数据缓冲区 */
    size_t                 bulk_in_size; /* 输入数据缓冲区的长度 */
    __u8                   bulk_in_endpointAddr; /* 块输入端点地址 */
    __u8                   bulk_out_endpointAddr; /* 块输出端点地址 */
    int                    errors; /* 保存最后发生的错误号 */
    int                    open_count; /* 设备打开者的数目 */
    spinlock_t             err_lock; /* 对访问 errors 进行同步 */
    struct kref             kref; /* 设备数据的引用计数 */
    struct mutex            io_mutex; /* 用于同步 */
};

/* 从 kref 的指针得到设备数据的指针 */
#define to_skel_dev(d) container_of(d, struct usb_skel, kref)

static struct usb_driver skel_driver; /* 驱动的前向声明 */
static void skel_draw_down(struct usb_skel *dev); /* 中止所有输出操作 */

/* 设备数据的引用计数减到 0 后的释放操作 */
static void skel_delete(struct kref *kref)
{
    struct usb_skel *dev = to_skel_dev(kref); /* 得到设备数据的指针 */
    usb_put_dev(dev->udev); /* 减少对所属 USB 设备的引用 */
    kfree(dev->bulk_in_buffer); /* 释放输入缓冲区 */
    kfree(dev); /* 释放设备数据自己 */
}

/* 类别设备的打开操作 */
static int skel_open(struct inode *inode, struct file *file)
{
    struct usb_skel *dev;
    struct usb_interface *interface;
    int subminor;
    int retval = 0;
    subminor = iminor(inode); /* 得到次设备号 */
    /* 查找次设备号对应的接口 */
    interface = usb_find_interface(&skel_driver, subminor);
    if (!interface) { /* 未找到 */
        err("%s - error, can't find device for minor %d", __func__, subminor);
        retval = -ENODEV;
        goto exit;
    }
    /* 得到指向设备数据的指针 */
    dev = usb_get_intfdata(interface);
    if (!dev) {
        retval = -ENODEV;
        goto exit;
    }
}

```



```

    }
    /* 增加对设备数据的引用计数 */
    kref_get(&dev->kref);
    mutex_lock(&dev->io_mutex); /* 与断开操作同步 */
    if (!dev->open_count++) { /* 判断是否首次打开，然后打开计数加 1 */
        retval = usb_autopm_get_interface(interface); /* 阻止自动挂起 */
        if (retval) { /* 如果失败则返回 */
            dev->open_count--;
            mutex_unlock(&dev->io_mutex);
            kref_put(&dev->kref, skel_delete);
            goto exit;
        }
    }
    /* 把以下代码的注释去掉可以使设备成为独占访问，只能同时打开一个 */
#ifdef 0
    else { /* 如果不是首次打开则返回错误，打开失败 */
        retval = -EBUSY;
        dev->open_count--;
        mutex_unlock(&dev->io_mutex);
        kref_put(&dev->kref, skel_delete);
        goto exit;
    }
#endif
    file->private_data = dev; /* 设置文件的私有数据 */
    mutex_unlock(&dev->io_mutex);
exit:
    return retval;
}

/* 类别设备的关闭操作 */
static int skel_release(struct inode *inode, struct file *file)
{
    struct usb_skel *dev;
    dev = (struct usb_skel *)file->private_data;
    if (dev == NULL) return -ENODEV;
    mutex_lock(&dev->io_mutex); /* 与断开操作同步 */
    if (!--dev->open_count && dev->interface)
        usb_autopm_put_interface(dev->interface); /* 允许自动挂起 */
    mutex_unlock(&dev->io_mutex);
    /* 减少对设备数据的引用计数 */
    kref_put(&dev->kref, skel_delete);
    return 0;
}

/* 类别设备的 flush 操作 */
static int skel_flush(struct file *file, fl_owner_t id)
{
    struct usb_skel *dev;
    int res;
    dev = (struct usb_skel *)file->private_data;
    if (dev == NULL) return -ENODEV;

```

```

    mutex_lock(&dev->io_mutex); /* 与输入输出操作同步 */
    skel_draw_down(dev); /* 中止所有输入输出操作 */
    /* 读出最后的错误码 */
    spin_lock_irq(&dev->err_lock); /* 对访问错误码进行同步 */
    res = dev->errors ? (dev->errors == -EPIPE ? -EPIPE : -EIO) : 0;
    dev->errors = 0; /* 清空错误码 */
    spin_unlock_irq(&dev->err_lock);
    mutex_unlock(&dev->io_mutex);
    return res; /* 返回错误码 */
}

/* 类别设备的读操作 */
static ssize_t skel_read(struct file *file, char *buffer,
                        size_t count, loff_t *ppos)
{
    struct usb_skel *dev;
    int retval;
    int bytes_read;
    dev = (struct usb_skel *)file->private_data;
    mutex_lock(&dev->io_mutex); /* 与断开操作同步 */
    if (!dev->interface) { /* 判断是否已断开 */
        retval = -ENODEV;
        goto exit;
    }
    /* 调用同步块输入接口 */
    retval = usb_bulk_msg(dev->udev, /* 指向 USB 设备 */
                          usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr), /* 管道号 */
                          dev->bulk_in_buffer, /* 输入缓冲区 */
                          min(dev->bulk_in_size, count), /* 要读的字节数 */
                          &bytes_read, /* 读到的字节数 */
                          10000 /* 超时限制 */
                          );
    if (!retval) { /* 如果成功 */
        /* 复制到用户态内存 */
        if (copy_to_user(buffer, dev->bulk_in_buffer, bytes_read))
            retval = -EFAULT;
        else
            retval = bytes_read; /* 读到的字节数 */
    }
exit:
    mutex_unlock(&dev->io_mutex);
    return retval;
}

/* 写操作中提交 URB 的完成函数 */
static void skel_write_bulk_callback(struct urb *urb)
{
    struct usb_skel *dev;
    dev = urb->context; /* 得到指向设备数据的指针 */
    if (urb->status) { /* 如果错误码不为 0 */
        if (!(urb->status == -ENOENT || /* 被 kill 或 poison */

```

```

        urb->status == -ECONNRESET || /* 被 unlink */
        urb->status == -ESHUTDOWN)) /* 已断开 */
/* 以上三种错误码属于正常情况, 不输出调试信息 */
err("%s - nonzero write bulk status received: %d",
    __func__, urb->status);
spin_lock(&dev->err_lock); /* 访问错误码时的同步 */
dev->errors = urb->status; /* 保存错误码 */
spin_unlock(&dev->err_lock);
}
/* 释放为 URB 分配的缓冲区 */
usb_buffer_free(urb->dev, urb->transfer_buffer_length,
    urb->transfer_buffer, urb->transfer_dma);
up(&dev->limit_sem); /* 完成一个 URB, 释放一个信号灯 */
}

/* 类别设备的写操作 */
static ssize_t skel_write(struct file *file, const char *user_buffer,
    size_t count, loff_t *ppos)
{
    struct usb_skel *dev;
    int retval = 0;
    struct urb *urb = NULL;
    char *buf = NULL;
    size_t writesize = min(count, (size_t)MAX_TRANSFER); /* 限制输出的长度 */
    dev = (struct usb_skel *)file->private_data; /* 得到指向设备数据的指针 */
    if (count == 0) goto exit; /* 输出长度为 0, 直接退出 */
    /* 获取信号灯以控制传输中的 URB 的个数 */
    if (down_interruptible(&dev->limit_sem)) {
        retval = -ERESTARTSYS;
        goto exit;
    }
    spin_lock_irq(&dev->err_lock); /* 对访问错误码进行同步 */
    /* 读取错误码 */
    if ((retval = dev->errors) < 0) {
        dev->errors = 0;
        retval = (retval == -EPIPE) ? retval : -EIO;
    }
    spin_unlock_irq(&dev->err_lock);
    if (retval < 0) goto error; /* 如果有错误则返回错误码 */
    /* 分配 URB */
    urb = usb_alloc_urb(0, GFP_KERNEL);
    if (!urb) {
        retval = -ENOMEM;
        goto error;
    }
    /* 为 URB 分配缓冲区 */
    buf = usb_buffer_alloc(dev->udev, writesize, GFP_KERNEL,
        &urb->transfer_dma);
    if (!buf) {
        retval = -ENOMEM;
        goto error;
    }

```

```

    }
    /* 从用户态内存复制数据到缓冲区 */
    if (copy_from_user(buf, user_buffer, writesize)) {
        retval = -EFAULT;
        goto error;
    }
    mutex_lock(&dev->io_mutex); /* 与断开操作同步 */
    if (!dev->interface) { /* 如果已断开 */
        mutex_unlock(&dev->io_mutex);
        retval = -ENODEV;
        goto error; /* 返回错误码 */
    }
    /* 初始化 URB */
    usb_fill_bulk_urb( /* 块类型 */
        urb, /* URB */
        dev->udev, /* USB 设备 */
        usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr), /* 管道号 */
        buf, writesize, /* 缓冲区和写入的字节数 */
        skel_write_bulk_callback, /* 回调函数 */
        dev /* 回调函数上下文 */
    );
    urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
    usb_anchor_urb(urb, &dev->submitted); /* 锚定 URB */
    retval = usb_submit_urb(urb, GFP_KERNEL); /* 提交 URB */
    mutex_unlock(&dev->io_mutex);
    if (retval) { /* 如果提交失败 */
        err("%s - failed submitting write urb, error %d", __func__, retval);
        goto error_unanchor;
    }
    /* 释放对 URB 的引用 */
    usb_free_urb(urb);
    return writesize;
error_unanchor:
    usb_unanchor_urb(urb);
error:
    if (urb) {
        usb_buffer_free(dev->udev, writesize, buf, urb->transfer_dma);
        usb_free_urb(urb);
    }
    up(&dev->limit_sem);
exit:
    return retval;
}

/* 文件操作 */
static const struct file_operations skel_fops = {
    .owner      = THIS_MODULE, /* 所属模块 */
    .read       = skel_read, /* 读操作 */
    .write      = skel_write, /* 写操作 */
    .open       = skel_open, /* 打开操作 */
    .release    = skel_release, /* 关闭操作 */

```

```

        .flush      = skel_flush, /* flush 操作 */
};

/* 类别驱动 */
static struct usb_class_driver skel_class = {
    .name =          "skel%d", /* 设备名称 */
    .fops =          &skel_fops, /* 文件操作 */
    .minor_base =    USB_SKEL_MINOR_BASE, /* 次设备号起始值 */
};

/* 探测操作 */
static int skel_probe(struct usb_interface *interface,
                     const struct usb_device_id *id)
{
    struct usb_skel *dev;
    struct usb_host_interface *iface_desc;
    struct usb_endpoint_descriptor *endpoint;
    size_t buffer_size;
    int i;
    int retval = -ENOMEM;
    /* 分配内存以保存设备数据 */
    dev = kzalloc(sizeof(*dev), GFP_KERNEL);
    if (!dev) {
        err("Out of memory");
        goto error;
    }
    /* 初始化设备数据 */
    kref_init(&dev->kref);
    sema_init(&dev->limit_sem, WRITES_IN_FLIGHT);
    mutex_init(&dev->io_mutex);
    spin_lock_init(&dev->err_lock);
    init_usb_anchor(&dev->submitted);
    /* 增加对所属 USB 设备的引用计数 */
    dev->udev = usb_get_dev(interface_to_usbdev(interface));
    /* 保存接口数据的指针 */
    dev->interface = interface;
    /* 从当前设置中查找一个块输入端点和一个块输出端点 */
    iface_desc = interface->cur_altsetting; /* 指向当前设置 */
    for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
        endpoint = &iface_desc->endpoint[i].desc; /* 指向第 i 个端点 */
        /* 查找块输入型端点 */
        if (!dev->bulk_in_endpointAddr && /* 如果尚未找到块输入端点 */
            usb_endpoint_is_bulk_in(endpoint)) { /* 且这个端点是块输入型 */
            /* 这就是要找的块输入型端点, 为其初始化设备数据 */
            /* 设置缓冲区的大小, 使之至少能够容纳一个最大的包 */
            buffer_size = le16_to_cpu(endpoint->wMaxPacketSize);
            dev->bulk_in_size = buffer_size;
            /* 保存端点地址备用 */
            dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
            /* 分配输入缓冲区 */
            dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);

```

```

        if (!dev->bulk_in_buffer) {
            err("Could not allocate bulk_in_buffer");
            goto error;
        }
    }
    /* 查找块输出型端点 */
    if (!dev->bulk_out_endpointAddr && /* 如果尚未找到块输出端点 */
        usb_endpoint_is_bulk_out(endpoint)) { /* 且这个端点是块输出型 */
        /* 这就是要找的块输出型端点, 保存其地址备用 */
        dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
    }
}
/* 如果没有找到块输入端点和块输出端点, 说明不是本驱动能够管理的设备 */
if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
    err("Could not find both bulk-in and bulk-out endpoints");
    goto error;
}
/* 将设备数据与接口绑定 */
usb_set_intfdata(interface, dev);
/* 注册 USB 类别设备 */
retval = usb_register_dev(interface, &skel_class);
if (retval) {
    err("Not able to get a minor for this device.");
    usb_set_intfdata(interface, NULL);
    goto error;
}
/* 显示调试信息, 让用户知道这个 USB 设备对应哪一个字符设备 */
dev_info(&interface->dev,
        "USB Skeleton device now attached to USBSkel-%d",
        interface->minor);
return 0;
error:
    if (dev)
        /* 减少对设备数据的引用计数, 这会自动释放分配的内存 */
        kref_put(&dev->kref, skel_delete);
    return retval;
}

/* 断开操作 */
static void skel_disconnect(struct usb_interface *interface)
{
    struct usb_skel *dev;
    int minor = interface->minor;
    dev = usb_get_intfdata(interface); /* 得到指向设备数据的指针 */
    usb_set_intfdata(interface, NULL); /* 将接口的私有数据设为空 */
    /* 注销类别设备 */
    usb_deregister_dev(interface, &skel_class);
    /* 阻止启动新的输入输出 */
    mutex_lock(&dev->io_mutex); /* 与类别设备的操作同步 */
    dev->interface = NULL;
    mutex_unlock(&dev->io_mutex);
}

```

```

/* 撤销所有锚定的 URB */
usb_kill_anchored_urbs(&dev->submitted);
/* 减少对设备数据的引用计数 */
kref_put(&dev->kref, skel_delete);
/* 输出调试信息, 表明设备已断开 */
dev_info(&interface->dev, "USB Skeleton #%d now disconnected", minor);
}

/* 中止所有输出操作 */
static void skel_draw_down(struct usb_skel *dev)
{
    int time;
    /* 等待所有锚定的 URB 完成 */
    time = usb_wait_anchor_empty_timeout(&dev->submitted, 1000);
    /* 超时后仍未完成则撤销所有锚定的 URB */
    if (!time) usb_kill_anchored_urbs(&dev->submitted);
}

/* 挂起操作 */
static int skel_suspend(struct usb_interface *intf, pm_message_t message)
{
    struct usb_skel *dev = usb_get_intfdata(intf); /* 得到指向设备数据的指针 */
    if (!dev) return 0; /* 为了安全而进行判断 */
    skel_draw_down(dev); /* 中止所有输出操作 */
    return 0;
}

/* 恢复操作 (无操作) */
static int skel_resume (struct usb_interface *intf)
{
    return 0;
}

/* 复位前操作 */
static int skel_pre_reset(struct usb_interface *intf)
{
    struct usb_skel *dev = usb_get_intfdata(intf);
    mutex_lock(&dev->io_mutex); /* 与类别设备的操作同步 */
    skel_draw_down(dev); /* 中止所有输出操作 */
    return 0;
}

/* 复位后操作 */
static int skel_post_reset(struct usb_interface *intf)
{
    struct usb_skel *dev = usb_get_intfdata(intf);
    dev->errors = -EPIPE; /* 将错误号设为管道已断开 */
    mutex_unlock(&dev->io_mutex); /* 对应复位前操作的 mutex_lock */
    return 0;
}

```



```
/* USB 设备驱动数据 */
static struct usb_driver skel_driver = {
    .name          = "skeleton", /* 设备名称 */
    .probe         = skel_probe, /* 探测操作 */
    .disconnect    = skel_disconnect, /* 移除操作 */
    .suspend       = skel_suspend, /* 挂起操作 */
    .resume        = skel_resume, /* 恢复操作 */
    .pre_reset     = skel_pre_reset, /* 复位前操作 */
    .post_reset    = skel_post_reset, /* 复位后操作 */
    .id_table      = skel_table, /* 匹配列表 */
    .supports_autosuspend = 1, /* 支持自动挂起 */
};

/* 模块的初始化函数 */
static int __init usb_skel_init(void)
{
    int result;
    /* 注册 USB 设备驱动 */
    result = usb_register(&skel_driver);
    if (result) err("usb_register failed. Error number %d", result);
    return result;
}

/* 模块的退出函数 */
static void __exit usb_skel_exit(void)
{
    /* 注销 USB 设备驱动 */
    usb_deregister(&skel_driver);
}

module_init(usb_skel_init);
module_exit(usb_skel_exit);

MODULE_LICENSE("GPL");
```



Part

第 5 部分 嵌入式 Linux 系统构建

第 17 章 Linux 内核构建

第 18 章 根文件系统构建

5

第 17 章 Linux 内核构建

随着 Linux 操作系统在嵌入式领域占据越来越主要的地位,对 Linux 内核定制和剪裁的需求越来越普遍。面对巨量的 Linux 内核源代码,开发人员需要针对自己的设备有选择地剪裁 Linux 内核,去掉不要的模块与功能支持,加入自己的驱动代码,进行特定的优化。这些工作都需要对 Linux 内核的配置编译系统有清楚的了解,懂得如何剪裁 Linux 内核为自己所需。

Linux 系统本身具备良好的模块化特性,可以简单地通过内核配置来达到剪裁模块并满足用户特定要求的目的,用户自行开发的内核代码也能够通过给 Linux 内核源码增加新的配置选项而融入内核。这个复杂的配置、编译维护系统就是 KBuild 系统。本章的主要内容就是讲述 KBuild 系统的结构和使用方法,具体来说,包含以下几个方面。

- ◆ 分析 Linux 内核源码中的配置系统结构。
- ◆ 如何向内核增加代码。
- ◆ 配置 Linux 内核的模块与功能,定制内核。

如无特别说明,本章所说的内核均以 Linux 2.6.30 版本为例,如果涉及到平台相关的代码,则均以 ARM 平台为例。

17.1 内核编译过程

Linux 内核所有版本的源码可从网址 <http://www.kernel.org> 下载得到。内核源码必须先进行配置才能编译。配置内核可采用以下几条命令之一:

```
make config # 命令行方式
make menuconfig # 命令行下的菜单方式
make xconfig # 图形界面方式,需 QT 支持
make gconfig # 图形界面方式,需 GTK 支持
```

然后就可以进行编译了:

```
make
```

这条命令实际上将编译得到内核的自解压映像 zImage 和所有的模块,也可以分别进行编译,如:

```
make zImage
make modules
```

如果系统中安装了 U-boot 的映像制作工具 mkimage,还可以直接编译出内核的 U-boot 映像,用如下命令:

```
make uImage
```

用以下命令可以将编译好的模块安装到系统中：

```
make modules_install
```

安装后的模块都放在 `/lib/modules` 目录下以内核版本为名称的目录中。

以上所说的都是本地编译内核的方法，如果要对内核进行交叉编译，则要注意环境变量的设置，比如配置内核时应使用如下命令：

```
make config ARCH=arm
```

这里变量 `ARCH` 指明了体系架构为 `ARM`。也可以将它设为环境变量，这样就不必每次都输入了。编译时还要指定所用的交叉编译器，如：

```
make ARCH=arm CROSS_COMPILE=arm-linux-
```

这里 `CROSS_COMPILE` 变量指明了所用的编译器，注意它的值只是编译器名称去掉 `gcc` 之后的前缀，这是因为在内核的 `Makefile` 中会自动给它加上 `gcc`。

安装模块时也可以指定安装的目录，如：

```
make modules_install INSTALL_MOD_PATH=${SYSROOT}
```

变量 `INSTALL_MOD_PATH` 用来指定安装目录。我们用 `SYSROOT` 变量表示预先定义好的根文件系统所在目录，这样安装后的模块放在 `${SYSROOT}/lib/modules` 目录下以内核版本为名称的目录中。

编译得到的内核映像都放在 `arch/${ARCH}/boot` 目录下，而内核模块一般与它的源码放在同一目录下。

用下面的命令可将源码清理干净，以便全部重新编译：

```
make clean
```

以上命令只清除编译结果。如果要将源码彻底清理干净，则可以用以下命令：

```
make distclean
```

这条命令会将配置文件一起删除，如果要重新编译，则必须重新配置。同时被删除的还有各个目录下由 `patch` 命令产生的 `*.orig` 文件和 `*.rej` 文件，以及 `ctags` 工具产生的 `tags` 文件。

内核的配置选项繁多，并且还在随版本的升高而不断增加。配置内核是一个相当繁琐而耗时的过程。配置的结果将保存为源码目录中的 `.config` 文件，这是一个隐藏文件。如果有已经配置好的配置文件，则直接将它复制到内核源码目录中并改名为 `.config` 即可。实际上，内核对于所支持的机器类型给了很多默认的配置文件的，放在源码的 `arch/${ARCH}/configs` 目录中，其中 `ARCH` 变量是体系架构类型。这些配置文件还可以通过如下命令发挥作用：

```
make s3c2410_defconfig ARCH=arm
```

这条命令将以 `arch/arm/configs/s3c2410_defconfig` 文件作为配置文件，它是内核源码中以

S3C2410 为处理器的机器类型的默认配置文件。

17.2 内核配置系统架构

在讨论 Linux 内核配置系统之前，首先来了解一下内核的源码结构。随着内核版本的升级，源码的目录也在不断地优化调整。这里以 Linux 2.6.30 内核为例，其源码目录中包含的主要目录如表 17.1 所示。

表 17.1 内核源码主要目录

目录名	包含内容
arch	体系架构相关的代码，与 ARM 相关的代码放在 arch/arm 目录下
Documentation	说明文档
drivers	驱动程序
fs	文件系统
init	内核初始化代码（不是引导的代码）
ipc	进程间通信
kernel	内核核心代码
lib	内核使用的库函数
mm	内存管理
net	网络协议
include	头文件

Linux 内核的配置系统大致可以分为三个部分，分别说明如下。

- ◆ **Makefile**：分布在 Linux 内核源码的各个目录下，定义了内核的编译规则。
- ◆ **Kconfig**：分布在 Linux 内核源码的各个目录下，定义了用户配置选项。
- ◆ **各种配置工具**：包括配置命令解释器（对配置脚本中使用的配置命令进行解释）和配置用户界面。

这些配置工具大都使用各种脚本语言编写，因此可以直接用文本编辑器修改。

17.2.1 内核 Makefile

Makefile 规定了内核的编译过程，包括以下主要步骤：首先根据配置的情况，构造出需要编译的源文件列表，然后分别编译，并把目标代码链接到一起，最终形成 Linux 内核的二进制文件。

17.2.1.1 内核 Makefile 的组织

由于 Linux 内核源代码是按照树形结构组织的，所以 **Makefile** 也分布在目录树的各个目录中。Linux 内核中的 **Makefile** 以及与 **Makefile** 直接相关的文件有如下几个。

- ◆ **Makefile**：顶层 **Makefile**，是整个内核配置、编译的总体控制文件。
- ◆ **.config**：内核配置文件，对内核进行配置后生成或更新。

- ◆ `arch/${ARCH}/Makefile`: 各种体系架构目录下的 `Makefile`, 如 `arch/arm/Makefile`, 这是针对特定体系架构的 `Makefile`。
- ◆ 其他各个子目录下的 `Makefile`: 比如 `drivers/Makefile`, 负责所在子目录下源代码的管理。
- ◆ `scripts/Makefile.*`: 各个 `Makefile` 的通用规则文件。

对内核源码的配置完成后, 将产生 `.config` 文件。编译时, 顶层 `Makefile` 读入 `.config` 中的配置选项。

系统由顶层 `Makefile` 控制生成 `vmlinux` 文件和各个内核模块。在顶层 `Makefile` 中, 包含了特定 CPU 体系架构下的 `Makefile`, 这个 `Makefile` 中包含了体系架构相关的信息。

顶层 `Makefile` 根据内核配置决定哪些子目录编译进内核, 然后递归进入到相关的各个子目录中, 分别调用位于这些子目录中的 `Makefile` 以生成目标。

位于各个子目录下的 `Makefile` 同样也根据 `.config` 文件给出的配置信息, 构造出当前目录下需要的源文件列表, 决定哪些文件编译进内核, 以及调用哪些下一级子目录中的 `Makefile`。

17.2.1.2 内核 Makefile 中的变量

顶层 `Makefile` 定义并向环境中导出了许多变量, 向各个子目录下的 `Makefile` 传递一些信息。有些变量, 比如 `core-y`, 不仅在顶层 `Makefile` 中定义并且赋初值, 而且在体系架构相关的 `Makefile` 中进行了扩充, 共同确定所应编译的文件。

下面将讲述内核 `Makefile` 中一些常用变量的含义。

一、obj-y

组织在 Linux 源码树中的源文件所在的目录都有一个 `Makefile` 文件 (非顶层 `Makefile`), 在其中使用 `obj-y` 变量定义需要编译进内核的文件, 也可以指定要包含进内核的下一级子目录。

`obj-y` 变量中定义的都是目标文件, 由当前目录下的 C 源文件 `*.c` 或汇编源文件 `*.S` 编译生成。这些目标文件将连同定义的下级目录中的 `built-in.o` 文件一起编译进当前目录的 `built-in.o` 目标文件中。

`obj-y` 变量中各个 `*.o` 文件的顺序是有意义的, 由 `__initcall` 定义的平级的初始化函数按照链接的顺序被调用。

`obj-y` 变量指定了一个目录下所有需要编译的目标文件 (由内核配置确定), KBuild 系统将 `obj-y` 列出的目标文件链接到一个单一的目标文件中, 这个文件就是 `built-in.o`。可以认为 Linux 源码的一个目录对应着一个 `built-in.o`。实际上, Linux 内核映像就是由这些与各个 Linux 源码目录一一对应的 `built-in.o` 链接而成的。

二、lib-y

`obj-y` 变量中的目标文件用于链接成 `built-in.o`, 目标文件也可能被链接到库文件中, 这由 `lib-y` 变量指定。所有罗列在 `lib-y` 变量中的目标文件都将被链接到该目录下的一个单一的库文件 `lib.a` 中。如果包含在变量 `lib-y` 中的目标文件也包含在 `obj-y` 变量中, 那么它就不会被链接进库文件。但如果包含在变量 `lib-y` 中的目标文件也包含在 `obj-m` 变量中, 它是要被链接进库文件的。

注意在同一个 `Makefile` 中既可以定义编译进 `built-in.o` 的文件, 同时也可以定义编译进 `lib.a` 的文件, 因此在同一个目录下既可以有 `built-in.o` 文件, 也可以有 `lib.a` 文件。

`lib-y` 变量通常只在 `lib` 目录和 `arch/${ARCH}/lib` 目录中使用。

三、`head-y`, `init-y`, `core-y`, `libs-y`, `drivers-y`, `net-y`
它们的含义分别解释如下。

- ◆ `head-y`: 链接内核映像时, 应最先链接的目标文件。
- ◆ `init-y`: 在 `head-y` 所指定的目标文件后链接的目标文件。
- ◆ `core-y`: 内核核心文件。
- ◆ `libs-y`: 库文件 `lib.a` 所在的路径。
- ◆ `drivers-y`: 驱动所在的目录下的文件。
- ◆ `net-y`: 内核网络模块。

这些变量都在顶层的 `Makefile` 下定义, 而在体系架构相关的 `Makefile` 中增加相应的一些目录。下面是从内核源码顶层 `Makefile` 中摘录的一些代码:

```
init-y      := $(patsubst %/, %/built-in.o, $(init-y))
core-y      := $(patsubst %/, %/built-in.o, $(core-y))
drivers-y   := $(patsubst %/, %/built-in.o, $(drivers-y))
net-y       := $(patsubst %/, %/built-in.o, $(net-y))
libs-y1     := $(patsubst %/, %/lib.a, $(libs-y))
libs-y2     := $(patsubst %/, %/built-in.o, $(libs-y))
libs-y      := $(libs-y1) $(libs-y2)

vmlinux-init := $(head-y) $(init-y)
vmlinux-main := $(core-y) $(libs-y) $(drivers-y) $(net-y)
vmlinux-all := $(vmlinux-init) $(vmlinux-main)
vmlinux-lds  := arch/$(SRCARCH)/kernel/vmlinux.lds
```

从这里可以看出内核编译时是如何处理这些变量的, 变量所包含的以斜杠 / 结尾的目录名将被替换为这个目录下的 `built-in.o` 或 `lib.a` 文件。最后构造了一系列 `vmlinux-*` 变量, 指出了 `vmlinux` 的各个组成部分, 这些变量将在链接生成 `vmlinux` 时用到。

四、`extra-y`

`extra-y` 变量定义了在当前目录下要编译但是不链接到 `built-in.o` 中的目标文件, 例如在 `arch/arm/kernel/Makefile` 文件中有如下定义:

```
extra-y := $(head-y) init_task.o vmlinux.lds
```

这里 `extra-y` 中的目标文件将被编译但不会链接到 `built-in.o` 中。之所以这样定义, 是因为上述列出的文件是应该首先链接的文件, 所以不要将其链接进当前目录下的 `built-in.o` 中, 而是在生成 `vmlinux` 时再单独链接。

17.2.1.3 创建 `vmlinux` 映像

以下是从内核源码的顶层 `Makefile` 中摘录的用于链接生成 `vmlinux` 映像的命令:

```
cmd_vmlinux__ ?= $(LD) $(LDFLAGS) $(LDFLAGS_vmlinux) -o $@ \
-T $(vmlinux-lds) $(vmlinux-init) \
--start-group $(vmlinux-main) --end-group \
$(filter-out $(vmlinux-lds) $(vmlinux-init) $(vmlinux-main) vmlinux.o
```

FORCE, \$^)

可以看出, `vmlinux-init` 变量指定的目标文件将会首先被链接, 其次是 `vmlinux-main` 变量所指定的文件, 而 `vmlinux` 目标所依赖的其他目标文件最后被链接, 使用的链接脚本则由 `vmlinux-lds` 变量指定。

链接的过程可由图 17.1 示意, 其中只列出了构成 `vmlinux` 映像的很少量的目标文件, 并且根据内核配置的不同, 图中的模块不一定会被链接进内核映像。

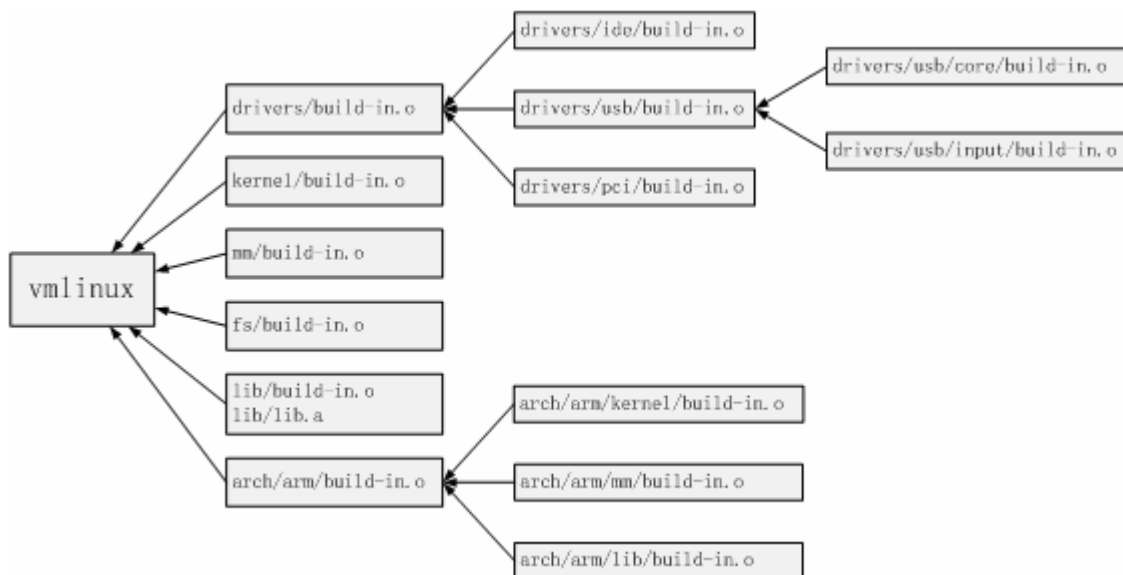


图 17.1 vmlinux 映像链接过程

17.2.1.4 创建可引导的内核映像

链接生成的 `vmlinux` 并不是一个可以直接在目标机上使用的二进制文件, 还需要经过一系列的处理步骤。一般的处理步骤可由图 17.2 示意。

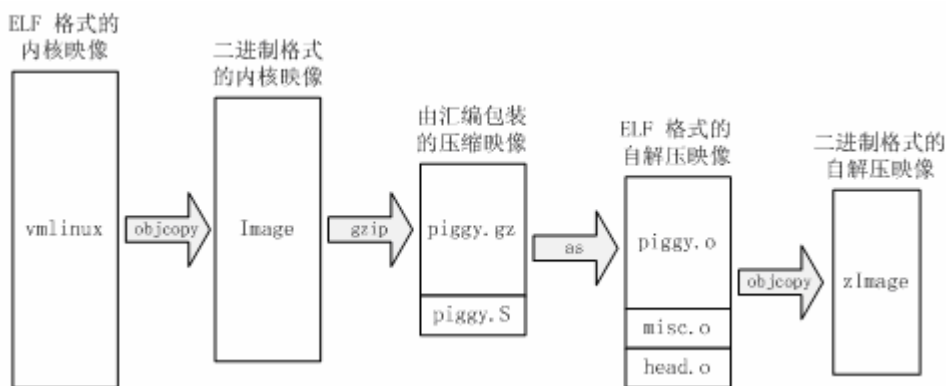


图 17.2 创建可引导内核映像的步骤

链接得到的 `vmlinux` 是 ELF 格式的目标文件, 需要经过 `objcopy` 的处理转化为二进制格式, 然后进行压缩, 产生 `piggy.gz` 文件。压缩后的内核映像显然已经不能执行, 它由文件 `piggy.gz`

中的一小段汇编代码包装成其他程序的数据段，进行汇编后得到 `piggy.o` 目标文件。`piggy.o` 将与解压缩代码及其他一些引导代码进行链接，得到一个 ELF 格式的自解压映像，最后再经过一次 `objcopy` 得到可直接引导的 `zImage` 自解压映像。

`zImage` 映像加载到目标机内存之后，从其最开始的第一条指令执行即可引导内核。引导时，首先将对包含在 `zImage` 内的压缩数据进行解压缩，得到未压缩的内核映像，然后跳转到其入口处执行。内核映像本身需要放在内存中固定的地址才能正常执行，但 `zImage` 的执行却不依赖于固定的内存地址，这是因为解压缩的代码本身是位置无关的，并且能够将解压后的内核映像放在预先确定的内存地址上，即使这个地址被 `zImage` 本身占据。

如果使用 U-boot 引导内核，可将 `zImage` 进一步包装为 U-boot 格式的映像文件。

17.2.2 KBuild 配置系统

内核源码树中几乎每个目录下都有两个文件 `Kconfig` 和 `Makefile`。这些 `Kconfig` 文件构成了一个分布式的内核配置数据库，每个 `Kconfig` 文件分别描述了所在目录源文件相关的内核配置菜单。当用户进行内核源码的配置时，`Kconfig` 的内容被读出并构造出配置选项以供选择，配置完成后配置选项的值被保存到 `.config` 文件中，内核的编译将根据这个文件的内容进行。

KBuild 配置系统的入口配置文件是 `arch/${ARCH}/Kconfig`。以下的内容摘录自文件 `arch/arm/Kconfig`，这是 ARM 体系架构的入口配置文件：

```
mainmenu "Linux Kernel Configuration"

config ARM
    bool
    default y
    select HAVE_AOUT
    select HAVE_IDE
    select RTC_LIB
    select SYS_SUPPORTS_APM_EMULATION
    select HAVE_OPROFILE
    select HAVE_ARCH_KGDB
    select HAVE_KPROBES if (!XIP_KERNEL)
    select HAVE_KRETPROBES if (HAVE_KPROBES)
    select HAVE_FUNCTION_TRACER if (!XIP_KERNEL)
    select HAVE_GENERIC_DMA_COHERENT
    help
        The ARM series is a line of low-power-consumption RISC chip designs
        licensed by ARM Ltd and targeted at embedded applications and
        handhelds such as the Compaq IPAQ. ARM-based PCs are no longer
        manufactured, but legacy ARM-based PC hardware remains popular in
        Europe. There is an ARM Linux project with a web page at
        <http://www.arm.linux.org.uk/>.

# 此处省略若干行……

source "init/Kconfig"

source "kernel/Kconfig.freezer"
```



```

menu "System Type"

choice
    prompt "ARM system type"
    default ARCH_VERSATILE

# 此处省略若干行.....

menu "Bus support"

config ARM_AMBA
    bool

# 此处省略若干行.....

menu "Kernel Features"

source "kernel/time/Kconfig"

config SMP
    bool "Symmetric Multi-Processing (EXPERIMENTAL)"
    depends on EXPERIMENTAL && (REALVIEW_EB_ARM11MP ||
        MACH_REALVIEW_PB11MP)
    select USE_GENERIC_SMP_HELPERS
    help
        This enables support for systems with more than one CPU. If you have
        a system with only one CPU, like most personal computers, say N. If
        you have a system with more than one CPU, say Y.

# 此处省略若干行.....

menu "Boot options"

# Compressed boot loader in ROM. Yes, we really want to ask about
# TEXT and BSS so we preserve their values in the config files.
config ZBOOT_ROM_TEXT
    hex "Compressed ROM boot loader base address"
    default "0"
    help
        The physical address at which the ROM-able zImage is to be
        placed in the target. Platforms which normally make use of
        ROM-able zImage formats normally set this to a suitable
        value in their defconfig file.

# 此处省略若干行.....

menu "CPU Power Management"

if (ARCH_SA1100 || ARCH_INTEGRATOR || ARCH_OMAP || ARCH_IMX || ARCH_PXA)

source "drivers/cpufreq/Kconfig"

# 此处省略若干行.....

```

```

menu "Floating point emulation"

comment "At least one emulation must be selected"

# 此处省略若干行……

menu "Userspace binary formats"

source "fs/Kconfig.binfmt"

# 此处省略若干行……

```

使用命令 `make menuconfig` 进行配置时将生成配置界面，这些配置界面反映的是配置数据库中的配置选项，以树形的方式组织，如：

```

+- Code maturity level options
| +- Prompt for development and/or incomplete code/drivers
+- General setup
| +- Networking support
| +- System V IPC
| +- BSD Process Accounting
| +- Sysctl support
+- Loadable module support
| +- Enable loadable module support
|   +- Set version information on all module symbols
|   +- Kernel module loader
+- ...

```

以上内容摘自内核源码中的 `Documentation/kbuild/kconfig-language.txt` 文件。

`Kconfig` 文件中的选项可以有依赖关系，这些依赖关系决定了选项是否可见。父选项可见，子选项才能可见。

如图 17.3 所示是 Linux 内核的配置菜单主界面，从中可以看出它与 `Kconfig` 文件中内容的对应关系。

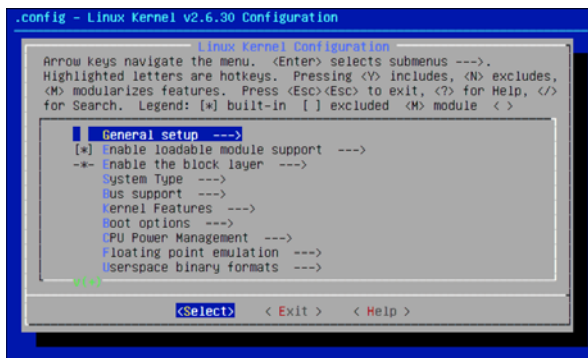


图 17.3 Linux 内核配置菜单界面

`Kconfig` 文件中的一些主要关键字的作用说明如表 17.2 所示。

表 17.2 Kconfig 文件中的关键字

关键字	作用
<code>mainmenu</code>	主菜单，整个 Kconfig 体系中只有一个
<code>config</code>	定义一个选项条目，对应于菜单中的一个选项
<code>menu</code>	定义一个菜单， <code>menu</code> 与 <code>endmenu</code> 之间可以是 <code>config</code> 选项，也可以嵌套子菜单
...	
<code>endmenu</code>	
<code>choice</code>	将多个选项条目组合在一起，让用户进行选择。 <code>choice</code> 与 <code>endchoice</code> 之间是 <code>config</code> 选项
...	
<code>endchoice</code>	
<code>sources</code>	将另一个文件的内容包含进来
<code>if</code>	代码分支， <code>if</code> 后面的条件成立时， <code>if</code> 与 <code>endif</code> 之间的内容才会生效
...	
<code>endif</code>	

简单地总结一下 Kconfig 文件的组织方式：首先整个 Kconfig 体系中只有一个主菜单，用 `mainmenu` 关键字定义。使用 `menu` 关键字可以创建菜单，菜单下可以包含配置的选项，也能够再包含菜单，这样就形成了树形的配置结构。

下面将对各个关键字的用法进行具体的说明。了解它们的用法之后，就不难将自己的配置内容增加到 Kconfig 文件中了。

17.2.2.1 config 选项条目

`config` 关键字用于定义新的选项条目，其定义的条目下面几行是该配置选项的属性。属性可以是该配置选项的类型、输入提示、依赖关系、帮助信息和默认值等。以下将分别说明 `config` 条目的属性设置。

一、类型定义和输入提示

类型定义为以下这些关键字之一。

- ◆ `bool`：逻辑型，有两个取值，`y` 或者 `n`。
- ◆ `tristate`：三态型，有三个取值，`y`，`m` 或者 `n`。
- ◆ `string`：字符串型，用户可以输入字符串。
- ◆ `hex`：十六进制型，用户可以输入十六进制数。
- ◆ `int`：整数型，用户可以输入十进制数。

类型定义后可以紧跟输入提示，如：

```
bool "Networking support"
```

也可以单独用 `prompt` 关键字指明输入提示，如上面这行语句可改写为：

```
bool
prompt "Networking support"
```

输入提示的完整格式如下：

```
prompt "提示内容" if 表达式
```

这里 **if** 及其后的部分是可选的，它表示一个依赖关系。每个菜单选项最多只能有一个显示给用户的输入提示。

二、默认值

选项的默认值用 **default** 语句设置，格式如下：

```
default 表达式 if 表达式
```

这里 **if** 及其后的部分是可选的，它表示一个依赖关系。一个配置选项可以有任意多个默认值。如果有多个默认值，那么只有第一个被定义的值是可用的。

如果可以显示输入提示的话，就会把默认值显示给用户，并可以让用户进行修改。如果用户没有设置，配置选项的值就是默认值。

三、依赖关系

选项的依赖关系由 **depends on** 语句设置，格式如下：

```
depends on 表达式
```

依赖关系会应用到选项的所有属性上，也就是说，类似下面的例子：

```
depends on BAR
bool "foo"
default y
```

将等价于以下的语句：

```
bool "foo" if BAR
default y if BAR
```

四、反向依赖关系

select 语句用于定义所谓的“反向依赖关系”，格式如下：

```
select 选项 if 表达式
```

这里 **if** 及其后的部分是可选的，它表示一个依赖关系。反向依赖关系只能用在 **bool** 型或 **tristate** 型选项上。它的含义可如下理解：如果当前选项的值为 **y**，则语句中指定的选项的值也被设为 **y**，总之被指定的选项的值一定要大于或等于当前选项的值。

五、数据范围

int 型和 **hex** 型的选项可以使用 **range** 语句限定输入值的范围，格式如下：

```
range 下限 上限 if 表达式
```

这里 **if** 及其后的部分是可选的，它表示一个依赖关系。限定之后，用户就只能输入大于等于下限、小于等于上限的值。

六、帮助信息

帮助信息的开始由以下语句标示：

```
Help
```

或者是以下语句：

```
---help---
```

两者是等价的。

帮助信息将结束在缩进比帮助信息的第一行小的行上。

17.2.2.2 菜单

`mainmenu` 表示顶层菜单，整个配置体系中只有一个，即：

```
mainmenu "Linux Kernel Configuration"
```

它将显示在配置菜单的最上方。

用 `menu` 语句可以增加子菜单，格式如下：

```
menu "提示语句"
    depends on 表达式
菜单内容
Endmenu
```

菜单本身也可以有依赖关系，并且它的依赖关系会应用到菜单内容的所有项目上。菜单内容可以是 `config` 选项条目或其他子菜单，也可以是提示性语句，如：

```
comment "At least one emulation must be selected"
```

这里 `comment` 语句中的提示信息将会着重显示在菜单内以提醒用户注意，但不产生任何配置项。

17.2.2.3 依赖关系

依赖关系决定了菜单、选项是否可见或者选项属性是否生效，这由一个表达式的结果决定。表达式的几种形式如表 17.3 所示。

表 17.3 依赖关系中的表达式

表达式的形式	结果
符号	如果符号是 <code>bool</code> 或 <code>tristate</code> 型的选项，它的值就是表达式的值，而其他类型的符号将导致表达式的值为 <code>n</code>
符号 = 符号	当两个符号相等时为 <code>y</code> ，否则为 <code>n</code>
符号 != 符号	当两个符号不等时为 <code>y</code> ，否则为 <code>n</code>
(表达式)	括号用于改变表达式求值的优先级顺序，值不变
! 表达式	$2 - \text{表达式}$
表达式 1 && 表达式 2	表达式 1 和表达式 2 的最小值
表达式 1 表达式 2	表达式 1 和表达式 2 的最大值

其中的符号可以是一个选项，也可以是一个字符串常数。表达式的取值是 `n`、`m` 和 `y`，为了

方便计算，可将其对应到数字 0、1 和 2。

当所依赖的表达式的值为 **m** 或 **y** 的时候，菜单或选项才是可见的，并且它的值不能超过所依赖的表达式的值。

17.2.2.4 选择项

使用 **choice** 语句可以把多个选项组合起来，格式如下：

```
choice
    属性设置
选择项内容
Endchoice
```

选择项的属性可以是类型、输入提示、默认值、依赖关系等，选择项的内容则可以是多个 **config** 选项条目。选择项只能为 **bool** 或者 **tristate** 类型。**bool** 类型的选择项只允许其内的一个选项条目设为 **y**，**tristate** 类型的选择项中虽然也只能有一个设为 **y** 的选项，但可以有多个设为 **m** 的选项。这主要是因为在在一个硬件有多个驱动的情况下，虽然最终只能有一个驱动被编译进内核，但所有的驱动都可以编译成模块。

选择项可以有一个属性 **optional**，它表示内部的所有选项都可以设为 **n**。

17.2.2.5 if 语句

if 语句的格式如下：

```
if 表达式
语句块
Endif
```

它的作用是把表达式所代表的依赖关系加在语句块中所有的内容上。

17.2.2.6 文件包含

文件包含使用 **source** 语句，格式如下：

```
source 路径
```

这里需要注意的是路径从 **Linux** 源码的顶层目录算起，而不是当前目录。

17.3 增加代码到内核

将自己开发的内核代码加入到 **Linux** 内核中，需要有三个步骤。首先，把自己开发的代码放入到内核源码树中合适的位置；其次，将开发的功能增加到 **Linux** 内核的配置选项中，以便用户可以通过配置系统选择此功能；最后，修改内核的 **Makefile**，根据用户选择将相应的代码编译进内核或编译成模块，当然，编译成模块时可以在内核源码树之外进行。

下面给出一个例子，将我们开发的 **HY2410A** 触摸屏驱动加入到内核源码中。

一、确定源码位置

内核中触摸屏的驱动都放在 `drivers/input/touchscreen` 目录下, 因此将我们的源码复制到这个目录下。

二、增加配置选项

打开文件 `drivers/input/touchscreen/Kconfig`, 可以看到这是一个大的选项菜单, 菜单中是各个驱动模块对应的选项。我们可以在菜单中增加自己的选项如下:

```
config TOUCHSCREEN_HY2410
    tristate "HY2410A touchscreen driver"
    help
        Say Y here if you have HY2410A touchscreen.
        If unsure, say N.
        To compile this driver as a module, choose M here: the
        module will be called hy2410_ts.ko
```

三、修改 Makefile

打开文件 `drivers/input/touchscreen/Makefile`, 可以看到其中多数是以下形式的语句:

```
obj-$(CONFIG_选项名) += xxxx.o
```

这里需要理解的一点就是 `Kconfig` 文件中的选项名对应着 `Makefile` 中的一个变量, 当这个选项的值设为 `y` 时, 上面的语句实际上展开为:

```
obj-y += xxxx.o
```

显然这条语句就可以将模块编译进内核。

因此我们可以在 `Makefile` 中增加如下一句:

```
obj-$(CONFIG_TOUCHSCREEN_HY2410) += hy2410_ts.o
```

通过上述几个步骤, 我们就可以将自己开发的触摸屏驱动加入到内核的配置与编译系统中, 通过菜单配置选项可以选择将驱动编译进内核、编译成模块或者不编译。

17.4 内核配置简介

编译一个内核, 最关键的是对内核系统进行满足自己需求的配置, 下面通过一个实例来具体说明如何对 Linux 内核进行配置。

Linux 内核支持众多的 CPU 架构, 如 `x86`, `ARM`, `mips` 等, 同一个架构下还有各种型号的开发板和机器。另外, 内核还支持各种文件系统、网络协议, 还有不计其数的驱动, 涉及到的技术概念很多, 并且随着内核版本的升级还在不断地增加新内容。因此对内核的配置特别复杂与繁琐。一般来说, 我们会从一个基本的配置入手, 将需要的配置加上, 去掉不需要的配置, 最终达到我们的要求。

下面我们对 Linux 2.6.30 内核的一些主要配置项进行说明, 以供参考。



一、General setup

基本配置项，可以配置内核的各种基本功能，如是否支持内存页交换、是否支持 IPC、是否支持消息队列等。

二、Enable loadable module support

是否支持动态加载模块，一般选择 `y`。

三、Enable the block layer

是否支持块设备层，一般选择 `y`。

四、System Type

这里选择所支持的机器类型。一个内核可以同时支持好几个机器类型，由 `bootloader` 引导时传入的参数进行选择。

五、Bus support

这里选择所支持的总线。

六、Kernel Features

内核特征配置，可以配置内存的分割方式、是否抢占、是否支持 EABI 等。

七、Boot options

与内核引导有关的配置，比如默认的引导参数、是否在 ROM 中原地执行等。

八、CPU Power Management

CPU 电源管理配置。

九、Floating point emulation

这里选择内核对浮点指令的模拟方式。当没有浮点协处理器时，软件中的浮点指令将触发未定义指令异常，内核通过软件的方法模拟运算结果返回。

十、Userspace binary formats

这里选择内核支持的应用程序格式。一般至少要支持 ELF 格式。

十一、Power management options

这里是电源管理选项。

十二、Networking support

这里是各种网络协议的选项。

十三、Device Drivers

这里是各种设备驱动的选项。应该说，这里是内核配置的重点。

如果嵌入式设备有网络支持，则应该尽早将网络调试通过，并配置好网络设备支持。通过挂载 NFS 网络文件系统，可以让系统先工作起来，以方便进一步的调试。

对于嵌入式系统来说，串口驱动几乎是必需的，因为我们用来调试的控制台建立在串口的基础上。

十四、File systems

这里是各种文件系统的支持选项。对于嵌入式开发来说，网络文件系统 NFS 几乎是必选项。



十五、Kernel hacking

这里是内核调试相关的选项。移植内核或开发驱动时，让内核多输出一些调试信息可能会很有帮助。这个菜单里的选项还可以让内核在引导的初始阶段就通过串口输出信息。

十六、Security options

这里是安全相关的选项。

十七、Cryptographic API

这里是加密算法相关的选项。

十八、Library routines

内核的函数库配置。这些函数内核本身并不使用，但如果某个动态加载的模块要用的话，也必须将这些函数库先加载进内核或编译进内核。

17.5 启动内核

内核编译之后的映像文件可以下载到目标机使用。如果是 U-boot，则需要使用内核的 `ulmage` 映像。以 HY2410A 开发板为例，在 U-boot 界面中，使用 `tftp` 方式下载内核映像的命令如下：

```
tftp 0x30080000 uImage
```

这条命令将把内核映像放在内存地址 `0x30080000` 处。然后用以下命令引导内核：

```
bootm 0x30080000
```

内核映像最终必须写入 Flash，以使目标机能够脱离主机启动。在 U-boot 界面内，先下载内核映像到目标机内存中，如：

```
tftp 0x30080000 uImage
```

然后擦除 Flash 上内核分区对应的区域，如：

```
nand erase 0x30000 0x300000
```

一般来说，Flash 的开始处放置的是 `bootloader` 本身，内核则在随后的分区上。最后将内存中的内核映像写入 Flash：

```
nand write 0x30080000 0x30000 $(filesize)
```

其中 `$(filesize)` 将替换为环境变量 `filesize` 的值，这一变量在文件下载结束后是所下载文件的大小。

内核映像写入 Flash 以后就不需要通过网络下载了，只需直接从 Flash 中读入内存，如：

```
nand read 0x30080000 0x30000 0x300000
```

然后就可以用 `bootm` 命令启动。



第 18 章 根文件系统构建

仅从文件系统类型以及存储文件的功能来讲，根文件系统与普通文件系统并没有本质上的区别，但它作为内核挂载的最顶层的文件系统有其自身的特点。首先它起到其他文件系统的“根”的作用，其次它要包含 Linux 系统初始化所需的目录和关键文件，否则即使内核成功启动也无法执行其他应用程序，无法与用户交互，整个系统也就无法使用。

本章将通过系统的初始化过程介绍根文件系统所需的关键文件，以及如何从无到有地创建一个根文件系统。

18.1 init 进程

下面将介绍 Linux 内核启动完毕，挂载根文件系统以后的初始化过程。了解系统初始化过程的目的是了解根文件系统中需要哪些关键的可执行文件及配置文件，以及它们之间的相互关系。

init 进程是内核启动完毕后执行的第一个进程。以下代码是内核启动最后阶段执行的内容，摘录自 Linux 2.6.30 内核源码的 init/main.c 文件的 init_post 函数中：

```
if (execute_command) {
    run_init_process(execute_command);
    printk(KERN_WARNING "Failed to execute %s. Attempting "
               "defaults...\n", execute_command);
}
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");
panic("No init found. Try passing init= option to kernel.");
```

在上述程序代码中，execute_command 是 bootloader 引导内核时传递给内核的参数 init 的值。从代码中可以看出，如果没有传递 init 参数给内核，则内核会顺次尝试执行 /sbin/init，/etc/init 等程序作为 init 进程。这些代码实际上是在 1 号进程中执行的，但在执行时它还只是一个内核进程。run_init_process 函数执行成功后不会返回，它将使当前进程转变为一个用户态进程。因此，init 进程可以说是从内核态到用户态的一座桥梁。

从用户态来看，init 进程是所有进程的共同祖先，其 PID 为 1。init 进程执行什么内容决定了整个系统是怎样初始化的。一般来说，它会根据相关的配置文件执行，决定启动哪些应用程序同时监控这些应用程序的运行，比如启动一个登录进程、打开一个 Shell 等。Shell 打开以后，我们就可以通过脚本或命令行界面启动其他的程序。

因此，作为一个可用的根文件系统，必须包含一个供 init 进程执行的可执行文件，可称为 init 文件。在嵌入式系统中，通常采用 busybox 软件包来提供这个文件。在默认情况下，这个文件被

安装在根目录下，文件名为 `linuxrc`，因此引导内核时需提供参数以便把 `init` 文件的路径通知内核，如在 U-boot 中设置下面的引导参数：

```
setenv bootargs 'console=ttySAC0 root=/dev/mtdblock2 rootfstype=yaffs2
init=/linuxrc'
```

最后的 `init=/linuxrc` 就是告诉内核 `init` 文件为 `/linuxrc`。

`linuxrc` 程序将根据配置文件 `/etc/inittab` 的内容执行，一个典型的 `inittab` 文件的内容如下：

```
::sysinit:/etc/init.d/rcS
::respawn:-/bin/sh
::restart:/sbin/init
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
```

注意这只是 `busybox` 所要求的配置文件格式，如果采用其他软件，则格式可能不同，甚至根本不需要这个配置文件。

配置文件的每一行格式如下：

```
tty:runlevel:action:process
```

其中各个字段的含义如下。

- ◆ `tty`：它代表执行程序 `process` 时所用的 `tty` 设备，内容就是 `/dev` 目录中的 `tty` 设备文件名，这个字段可省略。
- ◆ `runlevel`：`busybox` 不支持运行级别，所以这个字段完全被忽略。
- ◆ `action`：表示系统的某种动作。
- ◆ `process`：要执行的程序，可以有参数。

`action` 字段有以下这些可选值。

- ◆ `sysinit`：表示指定的程序要在系统初始化时执行。
- ◆ `respawn`：表示指定的程序要执行，并且进程一旦退出就会重新启动这个程序。
- ◆ `askfirst`：与 `respawn` 类似，只是在启动程序前要求用户按键确认。
- ◆ `ctrlaltdel`：当用户按下 `Ctrl+Alt+Del` 组合键时执行指定的程序。
- ◆ `restart`：当 `init` 进程重新启动时要执行指定的程序。
- ◆ `shutdown`：当系统关闭或重启前要执行指定的程序。
- ◆ `wait`：系统初始化时执行指定的程序，并且等待它执行完毕。
- ◆ `once`：系统初始化时执行指定的程序。

认识了 `inittab` 文件的格式之后，上述配置文件的作用可解读如下。

- ◆ 当系统初始化时，首先执行 `/etc/init.d/rcS`，它可以称为系统的初始化脚本。
- ◆ 初始化完毕后，启动 `/bin/sh` 作为 `Shell`，如果 `Shell` 意外退出则会重新启动。注意

/bin/sh 前面有一个减号，这表示它将以登录状态启动。

- ◆ 当用户按下 **Ctrl+Alt+Del** 组合键时，执行 **/sbin/reboot** 以重启系统。
- ◆ 当系统要关闭或重启前，执行 **/bin/umount -a -r** 以卸载所有文件系统。

Shell 启动以后，就可以与用户进行交互，启动其他各种程序，也可以使用 Shell 的启动脚本实现程序的自动运行。

18.2 创建根文件系统

本节将讲解如何从无到有地创建一个根文件系统。具体有如下几个步骤。

- step 1** 创建目录。
- step 2** 创建必要的设备文件。
- step 3** 安装所需的共享库。
- step 4** 安装 **init** 文件、Shell 和各种基本命令（可由 **busybox** 提供）。
- step 5** 创建和编辑配置文件。

18.2.1 创建目录

首先创建一个新的目录，作为整个根文件系统的根目录，如：

```
mkdir sysroot
```

这里创建了一个 **sysroot** 目录。当根文件系统制作完成后，可以将这个目录作为 **nfs** 服务的共享目录，这样目标机就能够通过网络挂载这个目录。

下面制作根文件系统的操作均对这个目录进行，所提到的绝对路径均以这个目录作为根目录，而不是主机自己的文件系统根目录。

如表 18.1 所示是一个根文件系统中所需的主要目录。

表 18.1 根文件系统中所需目录

目录	内容
/	根目录，其他目录都在这个目录下
/bin	存放基本命令的可执行文件
/sbin	存放基本系统管理类命令的可执行文件
/lib	存放基本共享库
/dev	存放设备文件
/etc	存放各种配置文件
/usr	包含若干子目录，用于存放用户程序和文档等
/usr/bin	存放用户程序的可执行文件
/usr/sbin	存放用户系统管理类程序的可执行文件
/usr/lib	存放用户程序所需的共享库
/proc	用于挂载 proc 文件系统
/sys	用于挂载 sysfs 文件系统
/tmp	存放临时文件

`/var`存放系统日志及一些服务程序的临时文件等

这些目录多数需要手动创建，其中有一些目录可在安装 **busybox** 时自动创建。一般来说，根文件系统内的主要目录和文件都会以 **root** 身份创建，因为将来挂载到目标机之后，文件的所有者和权限都保持不变，而这些文件在目标机上都是系统文件，应该是 **root** 所有的。

下文中将用环境变量 **SYSROOT** 表示根文件系统所在目录。

18.2.2 创建设备文件

Linux 下的设备文件按惯例都存放在 `/dev` 目录下，应用程序一般都需要通过设备文件来访问相应的设备。现在 **Linux** 系统上一般都采用 **udev** 来管理设备文件，这是一个专门用于管理设备文件的服务程序，它会根据 `/sys` 目录中的系统信息自动在 `/dev` 目录下创建或删除设备文件。

但是 **udev** 要产生作用，首先必须将 **sysfs** 文件系统挂载到 `/dev` 下，然后启动 **udev** 服务，这些工作一般在系统的初始化脚本中进行。然而系统初始化时要执行程序则必须有以下两个设备存在：`/dev/console` 和 `/dev/null`。这两个设备只能预先放在根文件系统中。

可用以下命令来创建这两个设备文件：

```
sudo mknod -m 600 ${SYSROOT}/dev/console c 5 1
sudo mknod -m 666 ${SYSROOT}/dev/null c 1 3
```

注意这些设备文件在主机上时实际指向主机的相应设备，但挂载到目标机之后，指向的就是目标机的设备。实际上也可以直接从主机的 `/dev` 目录下复制过来，如：

```
sudo cp -a /dev/console ${SYSROOT}/dev
sudo cp -a /dev/null ${SYSROOT}/dev
```

这里 **-a** 参数用来保留文件的所有属性，否则 **cp** 命令将试图从设备文件中读取内容并写入新文件。

在嵌入式系统中，出于某种需要可能不希望动态创建设备文件，这时，所需的设备文件就要用 **mknod** 命令预先在根文件系统中创建，或者将创建设备文件的命令写入系统的初始化脚本中。

18.2.3 安装共享库

一些基本的共享库必须放在根文件系统中，除非所有的程序都是静态链接的。所需的共享库都可以在交叉编译工具链的目录中找到，将其复制到根文件系统中即可，如：

```
cp -d libc.so.6 $(readlink libc.so.6) ${SYSROOT}/lib
```

这里 **-d** 参数的含义是如果文件是一个符号链接，则复制符号链接本身而不是它所指向的文件。**readlink** 则是一个 **Shell** 命令，它可以得到符号链接所指向的文件路径。

一般来说，一个共享库会对应着一个包含真正内容的文件，同时有若干个符号链接指向它，这样做是为了便于管理。在复制共享库的时候尽量保持这一模式，将应用程序所需的符号链接与包含真正内容的共享库文件一起复制过去。

如果要运行 **busybox**，则可能需要以下共享库：

```
libc.so.6
libm.so.6
ld-linux.so.2
libcrypt.so.1
```

当然，所需的共享库版本取决于编译应用程序时的工具链，因此我们推荐最好将编译程序的工具链中的共享库复制过去。

注意共享库之间也有相互调用的依赖关系，只有所需的共享库都存在，应用程序才能正常运行。用交叉编译工具链中的 `arm-linux-readelf` 工具可以查看 ARM 格式的 ELF 文件的信息，也可以得到它所依赖的共享库。实际上，如果仅仅查看动态链接信息，则主机上的 `readelf` 工具一样适用于 ARM 格式的文件，而且输出更为友好的信息，如：

```
readelf -d busybox
```

所输出的前几行信息如下：

```
Dynamic section at offset 0xb8100 contains 22 entries:
  Tag                Type                               Name/Value
 0x00000001 (NEEDED)             Shared library: [libcrypt.so.1]
 0x00000001 (NEEDED)             Shared library: [libm.so.6]
 0x00000001 (NEEDED)             Shared library: [libc.so.6]
```

从中可以清楚地看到它依赖于 `libcrypt.so.1`、`libm.so.6` 和 `libc.so.6`。

除以上基本库之外，还有以下一些其他的常用共享库：

```
libdl.so.2
libpthread.so.0
libutil.so.1
```

如果要使用域名解析相关的函数，则需要以下共享库：

```
libresolv.so.2
libnss_dns.so.2
```

如果要运行 C++ 源码编译得到的应用程序，则还需要以下共享库：

```
libstdc++.so.5
libgcc_s.so.1
```

18.2.4 安装 busybox

`busybox` 是嵌入式系统中常用的一个软件包，它把许多常用的 Linux 命令都集成到一个单一的可执行程序中，几乎只需要这一个可执行程序加上 Linux 内核就可以构建一个基本的 Linux 系统。`busybox` 忽略了许多不常用的功能，因此非常小巧，并且是完全模块化的，可以很容易地在编译时增加或删除其中包含的命令。

由于 `busybox` 具有这些特点，它非常适用于嵌入式 Linux 系统。下面将介绍 `busybox` 软件的配置、编译和安装。

一、获取源码

可用以下命令下载 busybox-1.10.2 的源码包：

```
wget http://www.busybox.net/downloads/busybox-1.10.2.tar.bz2
```

下载以后解压缩，将出现 busybox-1.10.2 源码目录，以下的配置和编译操作都在这个目录内进行。

二、配置

busybox 采用类似于内核的配置和编译系统，可用如下命令启动菜单界面进行配置：

```
make menuconfig
```

如图 18.1 所示为 busybox 菜单配置的主界面。

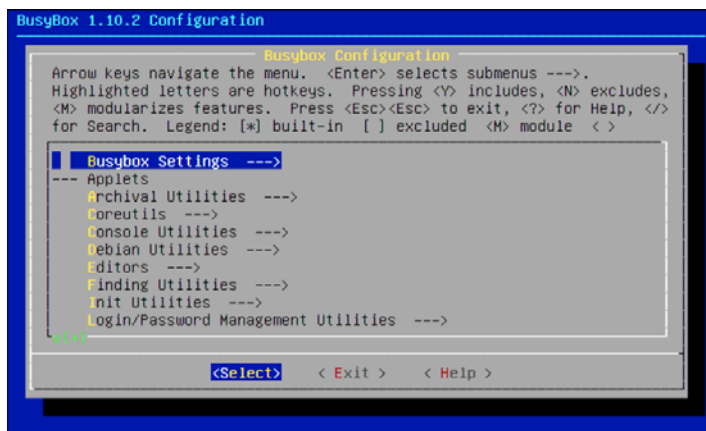


图 18.1 busybox 配置界面

其中 Busybox Settings 是编译 busybox 相关的配置，需要注意选择，按回车键进入后有如下子菜单：

```
General Configuration --->
Build Options --->
Debugging Options --->
Installation Options --->
Busybox Library Tuning --->
```

在 Build Options 菜单中有一个重要选项如下：

```
Build BusyBox as a static binary (no shared libs)
```

它表示是否以静态链接方式编译 busybox。如果选项被选中，则 busybox 将被编译成静态链接的可执行文件，运行时不需要其他共享库支持。由于我们的文件系统中安装了基本的共享库，故可以不选中这个选项。

在 Installation Options 中有一个重要选项如下：

```
BusyBox installation prefix
```

它表示 **busybox** 的安装目录，默认是 `./_install`，即源码目录下的 `_install` 目录。可将其设为根文件系统的根目录。这个选项也可以不改，在安装 **busybox** 时通过环境变量来指定安装目录。

主菜单中的其他子菜单是对命令的分类，其中可以配置所需的命令，可根据自己的需要进行选择。一般来说，如果对根文件系统的大小不是很苛求，可以直接使用 **busybox** 的默认配置，命令如下：

```
make defconfig
```

三、编译安装

配置完成后，可用以下命令进行编译：

```
make CROSS_COMPILE=arm-linux-
```

这里 `CROSS_COMPILE` 变量用于指定交叉编译工具链的前缀。

编译完成后可进行安装，命令如下：

```
make install CROSS_COMPILE=arm-linux- CONFIG_PREFIX=${SYSROOT}
```

这里 `CONFIG_PREFIX` 变量用于指定安装目录，如果不指定，则安装在配置时指定的安装目录中。安装过程中会自动创建所需的目录，包括 `/bin`、`/sbin`、`/usr/bin`、`/usr/sbin` 等，各种命令分别放在这些目录下，包括 `/bin/sh`，它是一个 Shell 程序，根目录下则生成一个 `linuxrc` 文件用于系统初始化。

实际上，只有 `/bin/busybox` 是一个真正的可执行文件，其他文件都是指向它的符号链接。**busybox** 通过命令行参数可以确定启动程序时所用的命令，从而去执行相应的功能。

18.2.5 创建配置文件

前面已经提及初始化程序 `linuxrc` 所需的 `inittab` 配置文件，不再介绍。在这个配置文件中指定了 `/etc/init.d/rcS` 作为初始化脚本，因此需要创建这个脚本文件。

下面是一个初始化脚本的例子：

```
#!/bin/sh
export PATH=/sbin:/bin:/usr/sbin:/usr/bin
mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t tmpfs mdev /dev
mount -t tmpfs tmp /tmp
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

需要提醒的是，这里首先设置了 `PATH` 环境变量，只是为了后续命令使用方便。由于执行初始化脚本与最后登录的并不是同一个 Shell，所以这里定义的环境变量不可能自动导入登录以后的 Shell。

下面将说明初始化脚本所做的几个重要工作。

一、挂载 `proc` 文件系统和 `sysfs` 文件系统

这个工作通过以下两条命令完成：


```
mount -t proc proc /proc
mount -t sysfs sysfs /sys
```

由于很多系统命令及应用程序靠访问 `/proc` 和 `/sys` 目录中的系统信息来实现功能，因此这两个文件系统必须挂载。特别是 `/sys` 目录，将马上在其后的命令中得到使用。

二、将 `/dev` 和 `/tmp` 挂载为内存文件系统

这个工作通过以下两条命令完成：

```
mount -t tmpfs mdev /dev
mount -t tmpfs tmp /tmp
```

这一步并不是必需的。但如果不这样做，动态创建和删除设备文件及临时文件的操作就会发生在真正的存储介质中，如 **Flash**、磁盘或网络文件系统中，效率会降低且完全没有必要。`tmpfs` 是内核提供的内存文件系统，它可以将内存作为文件系统来使用，其中的内容在关机后就全部消失。要使用 `tmpfs`，内核配置时的相关选项必须打开。

三、将 `mdev` 设为热插拔的监视程序

这个工作通过以下命令完成：

```
echo /sbin/mdev > /proc/sys/kernel/hotplug
```

`mdev` 是 `busybox` 提供的用于代替一般 PC 机上 `udev` 服务的程序。这样设置之后，系统中设备有热插拔事件时就会通知 `mdev` 程序，以动态创建或删除设备文件。

四、扫描并创建所有设备

这个工作通过以下命令完成：

```
mdev -s
```

这里 `-s` 参数表示让 `mdev` 这个程序扫描 `/sys` 目录中的内容并创建相应的设备文件。

`mdev` 程序需要一个配置文件 `/etc/mdev.conf`，这个文件的内容可以为空，但不能没有，因此可用以下命令创建：

```
touch ${SYSROOT}/etc/mdev.conf
```

其他的配置文件，如 `Shell` 的启动脚本 `/etc/profile` 等，可根据需要进行编辑。

18.3 挂载根文件系统

在上一节中，我们已经创建了一个基本的可以工作的根文件系统。本节将说明如何在目标机上使用创建好的根文件系统。

18.3.1 使用网络文件系统

使用网络文件系统可以让目标机把主机上共享的目录作为根文件系统，是用于调试的一种方便

的手段。使用网络文件系统必须有以下前提。

- ◆ 目标机支持网络。
- ◆ 目标机内核支持 NFS 文件系统。
- ◆ 主机上运行 nfs 服务，并共享根文件系统所在目录。

通过向内核传递引导参数可以指定内核使用 NFS 根文件系统。以 U-boot 为例，可设置引导参数如下：

```
setenv bootargs 'console=ttySAC0 root=nfs nfsroot=192.168.1.129:/home/jyg/work/sysroot ip=192.168.1.128 init=/linuxrc'
```

所用内核参数的含义解释如下。

- ◆ console：设置默认控制台所使用的设备。
- ◆ root：设置根文件系统挂载的设备。
- ◆ nfsroot：设置 NFS 根文件系统的源地址和目录。
- ◆ ip：设置目标机自身的 ip。
- ◆ init：设置启动后的初始化程序。

其中 nfsroot 的设置由服务器地址和源目录的绝对路径组成，中间用冒号隔开。

设置好引导参数后引导内核，内核启动结束后将挂载指定的目录作为根文件系统。此时可以在主机共享的相应目录内添加、删除、修改文件，所做的改变直接反映在目标机文件系统上。

18.3.2 使用 Flash 文件系统

嵌入式系统最终应用时需要把所用的文件系统固化在目标机非易失性存储介质内，一般来说是放在 Flash 中。在 Flash 上建立文件系统时，一般会采用专门为 Flash 设计的文件系统，如 JFFS（Journalling Flash File System，日志型 Flash 文件系统）。

Linux 2.6.30 内核支持 JFFS 的第二个版本。在将文件系统写入 Flash 之前，需要先在主机上制作文件系统的 jffs2 映像，这可以使用 mkfs.jffs2 命令来实现，如：

```
mkfs.jffs2 -n -s 512 -e 16KiB -d sysroot -o sysroot.jffs2
```

其中各个命令行参数的含义解释如下。

- ◆ -n：表示不要在每个擦除块上都加上清除标志。
- ◆ -s 512：表示每页大小为 512 字节。
- ◆ -e 16KiB：表示擦除块大小为 16KB。
- ◆ -d sysroot：表示根文件系统目录为 sysroot。
- ◆ -o sysroot.jffs2：表示输出的映像文件为 sysroot.jffs2。

其中的每页大小和擦除块大小应根据所用的 Flash 的参数做调整。

制作好映像文件以后，就可以将它写入 Flash。首先下载文件到目标机内存：

```
tftp 0x30000000 sysroot.jffs2
```

然后擦除 Flash 中文件分区对应的区域，并将内存中的映像写入：

```
nand erase 0x330000 0x3CD0000
nand write 0x30000000 0x330000 0x3CD0000
```

注意在 **bootloader** 阶段虽然没有 Flash 分区概念，但文件映像写入的位置必须与内核启动之后对 Flash 的分区相对应，否则无法挂载。HY2410A 开发板共拥有 64M 的 Flash，对 Flash 的分区是写在内核源码中的，如表 18.2 所示。

表 18.2 HY2410A 开发板 Flash 分区

编号	起始地址	大小	名称	用途	对应块设备
0	0	30000	bootloader	存放 U-boot	/dev/mtdblock0
1	30000	300000	kernel	存放内核映像	/dev/mtdblock1
2	330000	3CD0000	sysroot	存放根文件系统	/dev/mtdblock2

文件系统写入之后，我们就可以修改内核的引导参数，使其挂载 Flash 分区作为根文件系统，如：

```
setenv bootargs 'console=ttySAC0 root=/dev/mtdblock2 rootfstype=jffs2
init=/linuxrc'
```

其中 **root** 参数指定根文件系统所在的设备，**rootfstype** 指定根文件系统的类型。

使用以上方式固化文件系统有一个缺陷，如果文件系统对应的 Flash 分区内有坏块，则映像将无法写入，这对 NAND Flash 来说是常有的事。事实上，我们完全可以在系统启动之后再向 Flash 写入文件系统，对坏块的处理将由内核的文件系统驱动自动进行。

由于 Flash 上现在还没有文件系统，所以要先用 NFS 文件系统启动目标机。

在主机上将根文件系统打包，如：

```
tar -cjvf sysroot.tar.bz2 sysroot
```

这里 **sysroot** 就是根文件系统所在目录。然后将打包后的文件复制到根文件系统内：

```
sudo cp sysroot.tar.bz2 sysroot
```

以下在目标机的 Shell 界面内操作。首先在根目录下建立一个空目录 **sysroot** 并挂载相应的文件系统分区：

```
mkdir sysroot
mount -t yaffs /dev/mtdblock2 /sysroot
```

将打包好的根文件系统解压到所挂载的目录中：

```
tar -C / -xjvf sysroot-qt2.tar.bz2
```

因为根文件系统压缩包中包含了路径 **sysroot**，所以解压的目录指定到根目录就可以了。此时

文件系统已写入了 Flash，然后将设备卸载：

```
umount /sysroot
```

这样文件系统就被写入 Flash 的相应分区了。

当内核映像与文件系统全部写入 Flash 后，目标机就可以完全脱离主机而运行了。在 U-boot 界面中对 bootcmd 变量进行如下设置：

```
setenv bootcmd 'nand read 30080000 30000 300000; bootm 30080000'
```

内核启动参数设为从 Flash 挂载根文件系统。

目标机开机时，首先将启动 U-boot，倒计时结束后自动执行 bootcmd 中的命令。这些命令从 Flash 中将内核映像读入内存并执行。内核启动后，根据启动参数从 Flash 的文件系统分区挂载根文件系统，整个操作系统启动完毕。



Part

第 6 部分 应用编程

第 19 章 C++语言编程要点

第 20 章 嵌入式 GUI 编程

第 21 章 嵌入式数据库编程

第 22 章 产品开发实例：无线信息终端

6

第 19 章 C++ 语言编程要点

一方面, C++ 语言向下兼容 C 语言, 许多 C 语言程序可以不加修改地移植为 C++ 程序; 另一方面, C++ 语言良好地支持了面向对象的编程思想, 并且在语法上做了很多有益的扩展。由于这两个特点, 目前 C++ 语言已经成为各种大型应用软件及图形界面编程的首选语言。

本章的主要内容是对 C++ 语言的基本概念和用法进行介绍。由于 C 语言的语法规义几乎完全适用于 C++ 语言, 因此本章只介绍 C++ 特有的内容。实际上, C++ 语言在语法上的规定比 C 语言更加严格 (比如 C++ 语言规定 `main` 函数的返回值只能为 `int` 型、C++ 中的整数类型不能转换为枚举类型等), 故一些在 C 编译器下没有任何问题的程序用 C++ 编译器编译时却会有警告乃至错误。

本章适合对 C 语言有一定基础的读者快速跨入 C++ 编程的门槛, 以便在此基础上进一步学习图形界面应用程序的开发。

还有一点需要注意的是, 由于 C++ 语言比较复杂, 并且近年来其标准仍有所发展, 所以迄今为止没有完全兼容 C++ 标准的编译器问世。不同的编译器对一些语法细节的处理可能有所不同, 但这对一般的应用程序开发并不构成障碍。

在 C++ 语言中, 既可以使用 C 语言的 `/*` 和 `*/` 格式的注释, 也可以使用双斜杠 `//` 作为注释内容的开始, 两者均属于 C++ 标准。在本章中, 凡是 C++ 代码, 我们都采用双斜杠对代码进行注释。

19.1 布尔型数据

C++ 语言新增了一个基本数据类型——布尔型, 用 `bool` 定义, 用来表示只有“真”和“假”两个值的逻辑值, 如:

```
bool b = true; // 定义一个布尔型变量 b, 初始化为 true
if (b) b = false; // 将 b 的值改为 false
```

`true` 和 `false` 也是 C++ 关键字, 分别代表“真”和“假”。

C++ 语言规定比较操作符的返回值、逻辑操作符的操作数及 `if` 语句所判断的表达式必须是布尔型的。但由于布尔型与各种整数类型均可以自动转换, 所以类似 C 语言中用整型表示逻辑值的常用做法仍然有效。

19.2 引用

C++ 语言同样支持 C 语言的指针类型及 `*` 和 `&` 操作符, 但又给操作符 `&` 赋予了新的涵义, 即用来定义引用类型, 如:

```
int index = 1;
int &i = index; // 定义 i 是 index 变量的引用, 此后读写 i 如同读写 index
i = 2; // index 的值也变为 2
```

定义引用型变量时必须将其初始化为一个合法的变量, 否则是语法错误。

引用类型也可以用于定义函数参数, 此时在函数内读写该参数等价于读写调用函数时传入的相应实参, 如:

```
void swap(int &a, int &b) // 定义函数用于交换两个变量的值, 参数均为引用型
{
    int t;
    t = a; a = b; b = t; // 交换两个参数的值
}

int main(void)
{
    int a = 1, b = 2;
    swap(a, b); // 此后 a, b 的值真正被交换, a == 2, b == 1
    return 0;
}
```

函数的返回值也可以是引用类型, 这时要注意不能返回参数或局部变量本身, 因为它们的生存期到函数返回时结束, 函数调用者已无法使用。

引用类型也可以用 `const` 关键字修饰为只读型, 这时不能通过它去修改它所引用的变量, 如:

```
int index = 1;
const int &i = index; // 定义 i 是 index 的一个只读引用
i++; // 编译错误, i 不能被修改
```

对照指针类型和引用类型的用法, 可以发现引用实质上就是对指针的一种包装, 如表 19.1 所示。

表 19.1 引用与指针的对比

	C++	C
变量定义与访问	int index; int &i = index; i++;	int index; int *i = &index; (*i)++;
函数定义	void swap(int &a, int &b) { int t = a; a = b; b = t; }	void swap(int *a, int *b) { int t = *a; *a = *b; *b = t; }
函数调用	int a, b; swap(a, b);	int a, b; swap(&a, &b);

19.3 类和对象

类和对象是 C++ 语言的基本概念。与 C 语言相比较，类相当于类型的扩展，对象相当于变量的扩展。类用于定义对象，而类型用于定义变量，语法上具有相通之处。事实上，在 C++ 语言中，所有的数据类型都可以视为类。

19.3.1 类和对象的定义

定义类使用 `class` 关键字，举例如下：

```
class my_class { // 类的名称为 my_class
public: // 公共权限的成员可在任何地方访问
    my_class(); // 类的构造函数
    ~my_class(); // 类的析构函数
    int fun(int); // 公共成员函数
private: // 私有权限的成员只能在本类的成员函数中访问
    int n; // 私有成员变量
protected: // 保护权限的成员只能在本类及派生类中访问
    int m; // 保护成员变量
};
```



不要忘记定义完一个类最后要有分号。

定义类的语法与定义结构体类似，不同点之一在于，类里面不仅可以包含成员变量，还可以包含成员函数；不同点之二在于，类的成员有三种权限，分别说明如下。

- ◆ 公共权限：这种权限的成员可以在任何地方使用。
- ◆ 保护权限：这种权限的成员只能在本类及由本类继承的派生类的成员函数中访问。
- ◆ 私有权限：这种权限的成员只能在本类的成员函数中访问。

成员的访问权限是一种语法上的限制，仅在编译时有意义。

定义了类之后，就可以使用这个类来定义对象。与定义结构体变量不同，定义对象时类名前不需要有 `class` 关键字，如：

```
my_class obj; // 定义 my_class 类的对象 obj
```

19.3.2 构造与析构

类里面可以定义两个特殊的函数：构造函数和析构函数。构造函数的特点是必须与类的名称相同，它可以有参数，但不能定义返回值；析构函数的特点是它的名称为类名前加上符号 `~`，并且它既没有参数，也没有返回值。

构造函数将在对象生成时被调用。局部定义的对象将在程序执行到定义语句时生成，而全局定义的对象则在程序的主函数执行之前就生成。C++ 语言允许不在复合语句的开始处定义变量，如果复合语句中定义了多个对象，则它们按定义的顺序生成。

析构函数将在对象销毁时被调用。局部定义的对象将在程序执行到复合语句结束时被销毁，销毁的顺序与生成的顺序相反。全局定义的对象则在程序执行完毕之后被销毁。

构造函数的本义是让生成的对象在生成后能够自动初始化，而析构函数的本义是让对象在销毁前自动进行清理工作。一般来说，析构函数总是和构造函数执行相反的操作。

通常，构造函数和析构函数都声明为公共权限的成员函数，否则就不能随意定义对象，因为定义对象实际上意味着要调用构造函数。

如果没有定义构造函数与析构函数，则编译器会为所定义类自动生成构造函数和析构函数，这个构造函数没有参数，也不做任何操作。

如果构造函数有参数，则必须在定义变量时给出相应的参数，例如：

```
my_class obj(3); // 假定 my_class 的构造函数原型是 my_class(int i);
```

给出的实参将会在调用构造函数时传入。实际上，各种基本类型也可以用这种方法去初始化，例如：

```
int m(3), n(2); // 等价于 int m = 3, n = 2;
```

对象生成以后，就称它为相应的类的一个实例。

如果直接调用类的构造函数，则会生成一个临时对象，如：

```
my_class(); // 一个 my_class 类的临时对象
```

临时对象可以用在表达式中，但它在语句执行完毕时就会销毁。

19.3.3 类的实现

用 C++ 语言编程时，经常采用如下的方式：一个类就是一个模块，包括一个头文件和 C++ 源文件，在头文件中包含类的定义以方便其他模块使用，而在 C++ 源文件中定义类的成员函数。这时，类的成员函数是在类的外面定义的，有时将头文件中的内容称为类的接口，而源文件中的内容称为类的实现。仍以前述 `my_class` 类为例，它的一个实现如下：

```
my_class::my_class() // 构造函数
{
    m = 0; n = 0;
}

my_class::~~my_class() // 析构函数
{
}

int my_class::fun(int i) // 成员函数 fun
{
    return n = i;
}
```

注意在定义成员函数时，必须使用类名加域限定符 `::` 的方式来指明函数是属于某个类的。如果没有限定域，则编译器将认为所定义的成员函数是一个全局函数。

成员函数的定义也可以放在类里面，如：

```
class my_class {
public:
    my_class() { m = 0; n = 0; } // 构造函数
    virtual ~my_class() { } // 析构函数
    int fun(int i) { return n = i; } // 成员函数 fun
private:
    int n;
protected:
    int m;
};
```

一般来说，类的成员不能在类里面直接初始化，而应该在构造时初始化，这时构造函数可以书写如下：

```
my_class::my_class(): m(0), n(0) // 这里是初始化
{
    m = 0; n = 0; // 这里是赋值
}
```

成员变量（对象）的初始化放在构造函数的函数体之前，形式上是调用成员变量（对象）的构造函数。成员变量（对象）的构造在执行构造函数前完成。

19.3.4 访问对象

访问对象的数据成员的方法与访问结构体成员相同，类似的语法也用于访问对象的成员函数，例如：

```
my_class obj; // 定义 my_class 类的对象 obj
obj.fun(1); // 调用对象 obj 的成员函数 fun
my_class *pobj = &obj; // 定义对象指针指向 obj
pobj->fun(2); // 通过对象指针调用成员函数 fun
```

19.3.5 this 指针

this 是 C++ 关键字，它可以用在类的成员函数中，它的类型永远是指向所在类的一个指针，它的值则是对象自身的地址。例如在前述 **my_class** 类的成员函数 **fun** 中可以这样写：

```
int my_class::fun(int i) // 成员函数 fun
{
    return this->n = i; // this 指向对象自身，这里访问的是对象自己的成员 n
}
```

实际上，这里的 **this->** 完全可以省略，成员函数内如果出现成员变量名，则编译器默认它就是对象自己的成员。这时如果要访问一个同名的全局变量，则必须加上空的域限定符，如：

```
::n = 10; // 给全局变量 n 赋值
```

在调用成员函数时，本质上是有隐含的 **this** 指针作为参数传进去的，如：

```
obj.fun(1); // 在成员函数 fun 中, this 等于 &obj
pobj->fun(2); // 在成员函数 fun 中, this 等于 pobj
```

用 C 语言来模拟 C++ 的类将有助于对 **this** 指针的理解, 如表 19.2 所示。

表 19.2 类与结构体对比

C++		C	
类	<pre>class c { public: int n; void set_n(int); };</pre>	结构体	<pre>struct c { int n; };</pre>
成员函数	<pre>void c::set_n(int i) { n = i; }</pre>	函数	<pre>void c_set_n(struct c *this, int i) { this->n = i; }</pre>
对象定义	<code>c obj;</code>	变量定义	<code>struct c obj;</code>
调用成员函数	<code>obj.set_n(2);</code>	调用函数	<code>c_set_n(&obj, 2);</code>

19.3.6 new 和 delete

C++ 中定义了两个新的操作符用来动态生成和销毁对象。动态生成对象使用 **new** 操作符, 例如:

```
my_class *pobj = new my_class; // 生成新的 my_class 类的对象
```

new 操作符返回一个指向新对象的指针, 这样生成的对象的存储空间是动态申请的。同样, 生成对象时要调用类的构造函数, 如果构造函数有参数, 则使用 **new** 操作符时也要提供, 例如:

```
my_class *pobj = new my_class(3);
```

动态生成的对象必须用 **delete** 操作符销毁, 例如:

```
delete pobj; // 销毁对象时提供对象指针即可, 如果指针值为 0, 则无操作
```

同样, 用这种方法销毁对象也会自动调用其析构函数。

如果需要动态生成对象的数组, 则可以用 **new[]** 操作符, 如:

```
my_class *pobj = new my_class[5]; // 生成 5 个 my_class 类对象组成的数组
```

new[] 操作符返回指向数组的第一个元素的指针。需要注意的是, 如果类的构造函数有参数, 则不能使用 **new[]** 操作符。

对应的销毁对象数组的操作符是 **delete[]**, 例如:

```
delete[] pobj;
```



销毁对象数组时一定要用 `delete[]` 操作符，不要用 `delete`。

19.3.7 静态成员

类里面也可以用 `static` 关键字定义静态成员，如：

```
class my_class {
public:
    static int n; // 静态成员变量
    static void set_n(int i); // 静态成员函数
};
```

静态成员变量的特点是全局只有一个存储空间，通过类的任何实例访问到的都是同一个变量。实际上，即使类没有任何实例，也可以直接通过类名访问它的静态成员变量，如：

```
my_class::n = 2;
```

静态成员变量不能在构造函数处初始化，必须在类的外面用类似全局变量的方式进行初始化，如：

```
int my_class::n = 1;
```

这种初始化将在程序开始执行前进行一次。注意，这和在构造函数里赋值是不同的。如果在构造函数里赋值，则每次生成新对象时变量都会被赋值。

静态成员函数的特点是没有传递隐含的 `this` 指针参数，因此在函数体内不能访问非静态的成员。静态成员函数也可以直接通过类名来调用，如：

```
my_class::set_n(2);
```

19.3.8 只读成员

类的成员也可以被定义成只读的，这包括只读成员变量和只读成员函数两种情况。

一、只读成员变量

如果将类的成员变量加上 `const` 修饰符变成只读的，它就不可以被修改，因此，语法上规定只读的成员变量必须在构造时初始化，例如：

```
class my_class {
public:
    my_class(); // 构造函数
    const int n; // 只读成员变量
};

my_class(): n(2) // 如果这里不初始化，将出现语法错误
{
}
```

如果成员变量既是只读的又是静态的,那么就不要求它在构造时初始化,而是作为静态成员变量处理。实际上,这时是允许在类里面直接对成员变量进行初始化的,如:

```
class my_class {
public:
    static const int n = 2; // 静态只读成员变量,初始化为 2
};
```

二、只读成员函数

在 C++ 里, `const` 关键字还有一种新的用法,那就是加在成员函数的后面,如:

```
class my_class {
public:
    int n; // 成员变量 n
    int get_n() const; // 只读成员函数
};
```

把成员函数定义为只读的含义是它不会修改对象的成员变量,如在它的实现中:

```
int my_class::get_n() const
{
    n = 1; // 语法错误,不能修改成员变量
    return n; // 合法语句
}
```

对于一个只读的对象来说,调用它的只读成员函数是允许的,而调用非只读成员函数则是不允许的,例如:

```
my_class obj; // 定义一个对象
const my_class &o = obj; // 建立它的只读引用
o.get_n(); // 允许,如果 get_n() 不是只读成员函数则不允许
```

19.3.9 复制构造函数

类有一个特殊的构造函数称为复制构造函数,它可以根据一个对象去构造另外一个对象,它必须有一个类或者类的引用作为参数,如:

```
class my_int {
public:
    my_int(int i): p(new int(i)) { } // 构造函数
    my_int(const my_int &o): p(new int(*(o.p))) { } // 复制构造函数
    ~my_int() { delete p; } // 析构函数
private:
    int *p; // 用于保存一个整数的地址
};
```

这里将复制构造函数的参数定义为只读型,因为构造新对象时一般不需要对原对象进行修改。复制构造函数可以像一般的构造函数那样使用,如:

```
my_int a(1); // 生成对象 a
```

```
my_int b(a); // 以对象 a 为蓝本生成对象 b
```

实际上,复制构造函数往往是被隐含调用的。例如定义一个如下的函数:

```
void fun(my_int obj);
```

这里参数是类而不是类的引用,为了保证调用者提供的实参不被函数修改,编译器将隐含地调用复制构造函数生成一个实参的副本,函数内访问的其实是这个副本,这个副本在函数返回时销毁。由于直接以类作为参数将导致函数调用过程中生成和销毁对象,降低执行效率,所以对于复杂的类,一般会用它的引用类型作为参数。

如果不定义复制构造函数,则编译器会自动产生一个默认的复制构造函数,这个函数会将原对象的所有成员的值赋给新对象的对应成员。因此,如果从已有对象构造新对象时不能简单地复制成员变量的值,那么就需要自己定义复制构造函数。

例如在上述的 `my_int` 类中,成员指针 `p` 指向一个动态分配的整数,如果生成新对象时复制指针 `p` 的值,那么两个对象的成员 `p` 就指向相同的内存,两个对象最终都要析构,结果 `p` 指向的内存要被释放两次,逻辑上显然不正确。

19.3.10 友元

给类的成员赋予三种不同访问权限的目的是对数据进行更好的封装,以减少编程中容易产生的错误。在某些情况下,一个函数与一个类有非常密切的关系,需要访问类的私有成员变量,但又不可能声明为成员函数,这时可以将它声明为类的友元函数。

19.3.10.1 友元函数

声明友元函数使用 `friend` 关键字,如:

```
class my_class {
private:
    int m;
    friend void clear_my_class(my_class &); // 声明友元函数
};
```

在友元函数中可以访问这个类的任何成员,如:

```
void clear_my_class(my_class &obj)
{
    obj.m = 0;
}
```

19.3.10.2 友元类

友元的概念可以适用到类,称为友元类,如:

```
class my_class {
private:
    int m;
    friend class clear; // 声明友元类
}
```

```
};
```

在友元类的任何成员函数中都可以访问这个类的任何成员，如：

```
class clear {
public:
    clear(my_class obj) { obj.m = 0; }
};
```

19.4 类的继承

类的继承是 C++ 语言中重要的语法特征，也是实现面向对象编程的基础。

19.4.1 继承的语法

一个类可以继承另外一个类，语法如下：

```
class A { // 定义类 A
public:
    int n; // A 的成员变量
    void set_n(int i) { n = i; } // A 的成员函数
};

class B : public A { // 定义类 B 以公共方式继承类 A
};
```

这里，类 B 中虽然什么也没有定义，但由于它继承了类 A，所以相当于有了 A 的所有成员，例如：

```
B b; // 定义 B 类的对象 b
b.set_n(3); // 合法操作
```

这时称类 A 为类 B 的父类或基类，称类 B 为类 A 的子类或派生类。在子类的成员函数中可以访问父类的公共成员和保护成员，但不能访问私有成员。

子类的对象在生成时，首先自动调用父类的构造函数，然后自动调用子类自己的构造函数；子类的对象在销毁时，首先自动调用子类自己的析构函数，然后自动调用父类的析构函数。



通过子类对象调用父类的成员函数时，函数内的 this 指针的类型仍是指向父类的。

19.4.2 继承方式

类的继承有三种方式：公共方式、保护方式和私有方式，所用的关键字与定义类成员的权限时是相同的，它们的含义如下。

- ◆ 以公共方式继承时，外界对子类所拥有的父类成员的访问权限与父类一致。
- ◆ 以保护方式继承时，外界对子类所拥有的父类的公共成员的访问权限降格为保护权限，其

他权限成员的访问权限与父类一致。

◆ 以私有方式继承时，外界对子类所拥有的父类的所有成员的访问权限全部降格为私有权限。

如果继承时不指定方式，则默认是私有方式。

19.4.3 多重继承

一个类可以同时继承多个类，称为多重继承，例如：

```
class A { public: int a; }; // 定义类 A，拥有一个成员变量 a
class B { public: int b; }; // 定义类 B，拥有一个成员变量 b
class C: public A, public B { }; // 类 C 同时继承了类 A 和类 B
```

这样，类 C 就同时拥有了类 A 和 类 B 的所有成员。

在 C 语言中，当一个类型的指针转换为另外一个类型的指针时，它的值是保持不变的。但在 C++ 语言中，这个规律被打破了，考虑上述的多重继承情况，如果生成一个类 C 的实例并用指针访问它，如：

```
C *p = new C; // 生成类 C 的对象
p->a = 0; p->b = 0; // 通过类 C 的指针访问成员变量
((A *)p)->a = 1; // 将类 C 的指针转换成类 A 的指针后访问成员变量
((B *)p)->b = 2; // 将类 C 的指针转换成类 B 的指针后访问成员变量
```

显然，转换成 (A*) 型的指针只有指向类 A 的成员 a 才是有意义的，同样转换成 (B*) 型的指针只有指向类 B 的成员 b 才是有意义的，因此，即使它们是由同一个类 C 的指针转换而来的，也应该分别指向对象 (*p) 中属于类 A 的那一部分及属于类 B 的那一部分，可由图 19.1 解释。这里，指针的类型转换导致了指针值的变化。

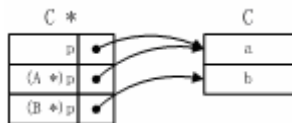


图 19.1 指针类型转换图解

一般来说，编译器在分配空间时，子类中属于父类的成员变量将放在子类自己的成员之前，如果同时继承多个父类，则父类的成员按顺序排列。将子类指针转换为父类指针时，编译器会自动把指针值加上父类成员相对于子类起始地址的偏移量。这种转换是安全的，并且有可能会自动发生。反之，将父类指针强制转换成子类指针则不一定安全。

19.5 函数和操作符重载

函数和操作符可以重载也是 C++ 语言的一大特色。利用函数重载可以定义多个同名但是不同原型的函数，它们互相不会冲突。操作符的重载突破了 C 语言中操作符的含义只能由编译器决定的局限性，编程者可以为自定义的数据类型重新诠释操作符的含义，进一步增强了语言的灵活性。

19.5.1 函数重载

19.5.1.1 全局函数

C++ 中允许定义函数名相同的多个函数，称为函数的重载。这时，为了区分各个函数，它们的参数列表必须是不同的，举例如下：

```
int add(int a, int b)    // 定义函数 add 用于两个整型数相加
{
    return a+b;
}

char add(char a, char b) // 定义函数 add 用于两个字符型数据相加
{
    return a+b;
}
```

这两个函数可以同时存在，不会产生重复定义的错误。调用函数时，编译器会根据参数的个数和类型自动选取匹配的函数，例如：

```
char c1 = 1, c2 = 2;
int n1 = 1, n2 = 2;
add(c1, c2); // 这里会调用 char add(char, char)
add(n1, n2); // 这里会调用 int add(int, int)
add(c1, n2); // 无法决定向哪个函数进行类型转换，发生“语义模糊”错误
```

在这里，如果相加的两个数据一个是字符型，一个是整型，则其中一个参数必须发生自动类型转换。由于存在两种转换的方案，最终会匹配到两个不同的函数上，因此编译器会给出“语义模糊”的错误。在使用 C++ 语言编程时，应尽量避免这种情况。

另外需要注意的一点是，根据函数调用的表达式无法区分传值和传引用两种情况。例如，在已经存在上述的 `add` 函数的情况下再定义下面的函数：

```
int add(int &a, int &b)
{
    return a+b;
}
```

这样，在进行两个整型数相加时同样会引起“语义模糊”错误。

19.5.1.2 成员函数

类的成员函数也可以重载，尤其是类的构造函数，因为名字是固定的，所以为了提供多种类的构造方法，只能进行重载，如：

```
class my_class {
public:
    my_class() { n = 0; } // 无参数的构造函数
    my_class(int i) { n = i; } // 有参数的构造函数
private:
```



```
int n;  
};
```

在一个类中，同名、同参数、同返回值的只读和非只读函数是可以同时存在的，也属于函数的重载，例如：

```
class my_class {  
public:  
    int get_n() const { return 1; } // 只读成员函数  
    int get_n() { return 2; } // 非只读成员函数  
};
```

调用时，如果对象是只读的，则调用只读版本，如果对象是非只读的，则调用非只读版本，如：

```
my_class obj1; // 定义一个对象  
const my_class &obj2 = obj1; // 定义对象的只读引用  
obj1.get_n(); // 调用非只读成员函数，返回 2  
obj2.get_n(); // 调用只读成员函数，返回 1
```

当然，即使某个成员函数只有只读版本，通过非只读类型的对象也是可以调用它的。

19.5.1.3 默认参数

C++ 语言支持函数的默认参数，举例如下：

```
int sum(int a = 1, int b = 2) // 参数 a 的默认值是 1, b 的默认值是 2  
{  
    return a+b;  
}
```

注意默认参数只能指定一次，一般把它放在函数的声明上，这时在函数的定义上就不能再指定。上述方式定义函数相当于同时定义了以下三个函数原型：

```
int sum(); // 省略的参数 a 初值为 1, b 初值为 2  
int sum(int a); // 省略的参数 b 初值为 2  
int sum(int a, int b); // 没有省略参数
```

因此调用时可以不指定默认参数，如：

```
sum(); // 返回 3  
sum(5); // 返回 7  
sum(8, 9); // 返回 17
```

使用默认参数的时候需要注意，只有当一个参数右边的所有参数都有了默认值以后，才能给它自己指定默认值，举一个错误的例子如下：

```
int sum(int a = 1, int b); // 错误
```

因为调用函数时在语法上没办法只写右边的参数而省略左边的参数。



19.5.2 操作符重载

19.5.2.1 基本语法

C++ 语言将操作符视为一种特殊的函数，并且允许对操作符的功能进行重新定义，称为操作符的重载。重载操作符使用 **operator** 关键字，例如：

```
class my_int {
public:
    my_int(): n(0) { } // my_int 类的构造函数
    my_int(int i): n(i) { } // my_int 类的构造函数
    int n; // 用于保存一个整数
};

// 定义加法操作符
const my_int operator+(const my_int &a, const my_int &b)
{
    return my_int(a.n+b.n); // 将两个参数的 n 成员相加并构造临时对象返回
}
```

加法操作符本身并不支持对两个 **my_int** 类的对象进行相加操作，但定义了上述加法操作符以后，就可以直接将两个 **my_int** 类的对象相加了，如：

```
my_int(1) + my_int(2); // 构造两个临时对象并相加
```

事实上完全可以将 **operator+** 整体视为一个函数，因此以上语句等价于：

```
operator+(my_int(1), my_int(2));
```

重载操作符时参数的个数必须与操作数的个数相等，类型没有特殊要求。一般来说，所定义的操作符应尽量模仿它本来的含义。比如在上述的例子中，加法操作通常不会改变操作数本身，因此两个参数都定义为只读的引用型，加法操作的结果通常与操作数是同类型的，并且不能作为左值使用，因此返回值定义为 **const my_int** 型。



重载全局的操作符时要求至少有一个参数是类，不能都是基本类型。

操作符也可以定义为类的成员，例如：

```
class my_int {
public:
    my_int(): n(0) { } // my_int 类的构造函数
    my_int(int i): n(i) { } // my_int 类的构造函数
    // 定义加法操作符，加法不改变自身，所以定义为只读成员函数
    const my_int operator+(const my_int &b) const
    {
        return my_int(n+b.n);
    }
    int n; // 用于保存一个整数
}
```

};

这里，加法操作符的第一个操作数将是对象自己，因此只需要一个参数，同样可以用以下语句进行相加：

```
my_int(1) + my_int(2); // 构造两个临时对象并相加
```

但是它等价的函数形式变为：

```
my_int(1).operator+(my_int(2)); // operator+ 是成员函数
```

19.5.2.2 赋值操作符

对于基本数据类型变量，赋值操作符的作用就是复制变量的值；对于对象来说，赋值操作符的默认操作就是复制所有成员的值。显然，这有时候是不符合要求的，因此需要重载赋值操作符，举例如下：

```
class my_int {
public:
    my_int(int i): p(new int(i)) { } // 构造函数
    my_int(const my_int &o): p(new int(*(o.p))) { } // 复制构造函数
    ~my_int() { delete p; } // 析构函数
    // 赋值操作符的定义
    my_int &operator=(const my_int &obj) {
        *p = *(obj.p); // 复制指针指向的整数的值，而不是简单地复制指针的值
        return *this; // 返回对象自身
    }
private:
    int *p; // 用于保存一个整数的地址
};
```

因为赋值操作符不会改变第二个操作数，所以将参数定义为只读的引用类型；又因为赋值操作符有返回值，故将返回值定义为对象自身的引用。



在 C++ 语言中，赋值操作符的返回值是可以作为左值使用的，这一点是与 C 语言不同的。

需要注意的是，并非代码中出现等号的地方就一定会调用赋值操作符，请看以下代码：

```
my_int a(1), b(1);
my_int c = a; // 这里是初始化，等价于 my_int c(a)，调用复制构造函数
b = a; // 这里是赋值操作，调用赋值操作符
```

也就是说，等号用在对象定义时表示进行初始化，实际上将调用复制构造函数而不是赋值操作符。

19.5.2.3 类型转换操作符

类型转换操作符也可以被重载，举例如下：

```
class type_test {
```

```
public:
    operator short() {return 1; } // 转换为 short 型
    operator int() {return 2; } // 转换为 int 型
    operator long() {return 3; } // 转换为 long 型
};
```

这里为一个类定义了三个类型转换操作符，并返回了不同的值。使用时结果如下：

```
type_test t;
short(t); // 结果为 1, C++ 中可以用函数形式进行类型转换
int(t); // 结果为 2
(long)t; // 结果为 3, 也可以沿用 C 语言的方式进行类型转换
```



定义类型转换操作符时不需要再写返回值的类型，因为函数名就已经说明了返回值的类型。

类型转换操作符的一个重要作用就是可以进行自动类型转换，仍以上述 `type_test` 类为例进行说明。首先定义一个函数：

```
int add(int a, int b) // 定义一个函数，用于两个整型数相加
{
    return a+b;
}
```

用这个函数对两个 `type_test` 类的临时对象进行操作：

```
add(type_test(), type_test()); // 对象自动转为整型，结果是 4
```

用加法操作符对两个 `type_test` 类的临时对象进行相加：

```
type_test() + type_test(); // 错误，语义模糊，无法决定转换成何种类型
```

类型转换操作符不仅可以将类转换为各种基本类型，也可以转换为其他类。定义类型转换操作符时需要特别注意的是避免产生语义模糊，如在上面的例子中对两个 `type_test` 类的对象使用加法操作符时，由于加法操作符本身就支持各种整数类型，所以导致编译器在多种可行的转换方案间无所适从。

除了类型转换操作符外，带有一个参数的构造函数也有类型转换的作用，举例如下：

```
class my_int {
public:
    my_int(int i) { n = i; } // 带一个整型参数的构造函数
    int n;
};

int get_my_int(const my_int &obj) // 以 my_int 类对象作为参数的函数
{
    return obj.n;
}
```

然后就可以直接将整型数作为参数调用 `get_my_int` 函数，如：

```
get_my_int(2); // 结果是 2，自动以 2 作为参数构造 my_int 类的临时对象
```

这种自动构造临时对象的功能有时候会给代码带来难以发现的问题。C++ 提供了一个关键字 `explicit` 可以禁止自动构造对象，举例如下：

```
class my_int {  
public:  
    explicit my_int(int i) { n = i; } // 带一个整型参数的构造函数  
    int n;  
};
```

这样，在需要构造对象的时候就必须明确地写出来，如：

```
get_my_int(my_int(2)); // 结果是 2
```

19.6 覆盖与虚函数

覆盖指的是在子类中定义与父类中成员函数同名同参数的函数。从本质上来说，这也可以视为一种重载，因为两者的成员函数有一个隐含的 `this` 指针参数是不同类型的。由覆盖而引申出来的虚函数的用法也是 C++ 语言用来支持面向对象编程的一个基本特征。

19.6.1 覆盖

在子类中可以定义与父类中同名的函数，举例如下：

```
class A {  
public:  
    char name() { return 'A'; } // 类 A 的 name 函数  
};  
  
class B: public A { // 类 B 继承类 A  
public:  
    char name() { return 'B'; } // 类 B 重新定义了 name 函数  
};
```

这样对类 B 的对象调用 `name` 函数时，实际调用的是类 B 中定义的函数而不是类 A 中定义的函数，这就是函数的覆盖，例如：

```
A a; B b; // a 是类 A 的实例，b 是类 B 的实例  
a.name(); // 实际调用类 A 的 name 函数，返回 'A'  
b.name(); // 实际调用类 B 的 name 函数，返回 'B'  
((A *)&b)->name(); // 实际调用类 A 的 name 函数，返回 'A'
```

19.6.2 虚函数和多态

在上述函数覆盖的例子中，虽然子类中已经重新定义了父类中的同名函数，但是当把指向子类

对象的指针强制转换为父类的指针时，调用的还是父类中的函数，但是很多时候希望在这种情况下调用的也是子类中的函数，这时就要使用虚函数。

声明虚函数使用 **virtual** 关键字，例如：

```
class A {
public:
    virtual char name() { return 'A'; } // 类 A 的 name 函数，是虚函数
};

class B: public A { // 类 B 继承类 A
public:
    char name() { return 'B'; } // 类 B 重新定义了 name 函数
};
```

在父类中声明的虚函数在子类中覆盖后还是虚函数，不需要再声明，即使从子类再继承出新类，这个函数仍是虚函数。但子类中声明为虚函数并不回溯决定父类中的同名函数也是虚函数。

使用虚函数以后，结果就发生了变化，如：

```
A a; B b; // a 是类 A 的实例，b 是类 B 的实例
a.name(); // 实际调用类 A 的 name 函数，返回 'A'
b.name(); // 实际调用类 B 的 name 函数，返回 'B'
((A *)&b)->name(); // 实际调用类 B 的 name 函数，返回 'B'
```

使用虚函数的目的在于，父类中无须知道虚函数在子类中的实现细节就可以基于虚函数实现其他功能，例如：

```
class A {
public:
    virtual char name() { return 'A'; } // 虚函数 name
    char lower_name() { // 将 name 函数返回的结果转换成小写字母
        return name()-'A'+'a';
    }
};

class B: public A { // 类 B 继承类 A
public:
    char name() { return 'B'; } // 类 B 重新定义了 name 函数
};
```

使用时会有如下结果：

```
A a; B b; // a 是类 A 的实例，b 是类 B 的实例
a.lower_name(); // 返回 'a'
b.lower_name(); // 返回 'b'
```

这里在类 A 内利用虚函数实现了一个新函数 `lower_name`，可以将 `name` 函数的结果转换成小写字母返回。当类 B 继承类 A 时，只要覆盖函数 `name`，`lower_name` 函数的返回结果也随之而变。如果 `name` 不是虚函数，那么即使通过类 B 的对象调用 `lower_name` 函数，函数内的 `this` 指针也是指向类 A 的，因此调用的也是类 A 自己的成员函数。

在上面的例子中，某个函数的功能在定义时尚不能完全确定，取决于继承它的子类如何覆盖这个函数，这种现象称为多态。多态是面向对象编程的基本特征之一。



如果一个类要被继承，那么它的析构函数通常声明为虚函数，这样才能保证当它的子类对象析构时总能调用到对象本身的析构函数。

19.6.3 虚函数的实现

调用一个非虚函数时，根据对象的类型，编译器完全可以决定具体链接到哪个函数，程序运行时，调用者与被调用函数的链接关系是不变的，因此，这种链接的方式称为静态绑定。与此相对的是用于虚函数的动态绑定方式，在编译阶段，调用虚函数时最终链接到哪一个函数是不确定的，只有到了运行阶段，才能根据生成的是何种对象来决定调用目标。

虚函数的动态绑定由虚函数表 (VTABLE) 和虚指针 (VPTR) 来实现。编译器为每个包含虚成员函数的类生成一个数组，用来保存各个虚函数的入口地址，这就是虚函数表；同时还为它们增加一个隐含的成员，称为虚指针，这个指针在对象构造时指向相应类的虚函数表。对虚成员函数的调用是这样进行的：首先从对象中得到虚指针的值，找到对应的虚函数表，然后通过查表得到虚函数的入口地址进行调用。由于子类对象的虚指针已在构造时指向子类的虚函数表，所以，即使把它作为父类使用，调用的虚函数还是子类自己的成员函数。类的虚指针和虚函数表的关系如图 19.2 所示。

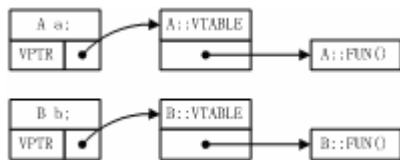


图 19.2 类的虚指针与虚函数表的关系

19.6.4 纯虚函数与抽象类

因为对虚函数的调用不依赖于它的具体实现，所以在 C++ 里允许在一个类里面声明没有实现的虚函数，称为纯虚函数。包含纯虚函数的类不能直接定义对象，但可以被其他类继承，这种类称为抽象类。只有当继承的类里面实现了父类所有的纯虚函数以后，它才可以用来生成对象，例如：

```
class A {
public:
    virtual char name() = 0; // 纯虚函数的声明，这个函数不需要实现
    char lower_name() {
        return name() - 'A' + 'a'; // 即使纯虚函数没有实现也可以被调用
    }
};

class B: public A {
public:
    char name() { return 'B'; } // 实现了父类的纯虚函数
};
```

在虚函数的声明后面加上 = 0 就可以将它声明为纯虚函数。以上的类在使用时有如下结果：


```
A a; // 错误, 抽象类不能用来定义对象
B b;
b.lower_name(); // 返回 'b'
```

19.7 名字空间

为了解决在大型项目中标识符名称的冲突问题, C++ 语言中提出了名字空间的概念。可以将变量、函数及类的声明或定义放在某个名字空间中, 不同的名字空间中即使有相同的名称, 它们指代的也不是同一个东西。名字空间的用法举例如下:

```
namespace foo { // 名字空间 foo
    int m = 1; // 全局变量 m, 初始化为 1
    class my_class { // 类
    public:
        explicit my_class(int i): n(i) { } // 将整数的值保存在私有变量中
    private:
        int n; // 用于保存一个整数
        friend int fun(const my_class &); // 声明为友元, 使函数能够访问私有变量
    };
    int fun(const my_class &obj) { // 函数
        return obj.n; // 返回对象中保存的整数值
    }
} // 这里不需要分号

namespace bar { // 名字空间 bar
    int m = 2; // 全局变量 m, 初始化为 2
    // 以下内容与名字空间 foo 中相同, 故省略
} // 这里不需要分号
```

这里定义了两个名字空间 `foo` 和 `bar`, 在它们内部定义了相同名字的变量、类和函数, 这些不会构成冲突, 但在使用时需要指明所在的名字空间, 如:

```
foo::fun(foo::my_class(foo::m)); // 返回 1
```

指明所在名字空间与指明所在类用相同的域限定符。如果对用到的每个标识符都要指明名字空间, 则稍显累赘且不易读, 这时可以用以下的方法预先声明所用的名字空间:

```
using namespace bar; // 声明使用名字空间 bar
fun(my_class(m)); // 这里的变量、函数、类都从名字空间 bar 中查找, 返回 2
```

这时要注意, 如果同时使用的多个名字空间 (以及不在名字空间) 中有相同的标识符, 则使用这个标识符时还是要指明所在的名字空间, 否则会有语义模糊的错误。

类的名称可以视为一个天然的名字空间, 因此 C++ 语言允许在类里面嵌套定义类型、枚举型的值甚至其他类, 它们与类外面定义的同名符号不相互冲突, 如:

```
class my_class {
public:
    class A { };
```

```
typedef int my_int;
enum my_enum {ZERO, ONE, TWO};
};
```

与名字空间不同的是，类里面定义的类型受访问权限的控制，并且对于类名也不能使用 `using namespace` 语句。

19.8 模板

C++ 语言中提出了模板的用法，可以让编译器自动“写”出新的函数或类。恰当地使用模板能够减少代码量，增强代码的可重用性。

19.8.1 模板函数

举个例子，如果要写一个函数用于交换两个变量的值：

```
void swap(int &a, int &b) // 交换两个整型变量的值
{
    int t;
    t = a; a = b; b = t;
}
```

这个函数只能用于两个整型变量值的交换，如果要交换其他类型的变量，则还要针对这个类型写新的函数。实际上，可以用模板的方法来实现一个通用的函数，如：

```
template <class T> // 声明模板，T 为模板参数，class 表示 T 为任意类
void swap(T &a, T &b) {
    T t;
    t = a; a = b; b = t;
}
```

这样就定义了一个模板函数。模板函数中可以用类型 `T` 定义变量，而 `T` 本身是模板参数，代表任意的类。这样，不管使用时 `T` 是什么类型，只要它支持赋值操作符，以上模板函数就是可用的，如：

```
char a = '1', b = '2';
swap(a, b); // 需要参数为 char 型的函数，T 自动代换为 char 实现新函数
```

这样，在调用模板函数时，编译器根据所需类型自动“写”了一个函数，如此就实现了函数对各种类型的通用。

模板的实现可由编译器自动选定，也可以明确指定，比如：

```
short a = 1, b = 2;
swap<short>(a, b); // 明确指定将 T 代换为 short
```

模板参数不仅可以是某个类型，也可以是某种类型的常数，比如：

```
template <char ch> // 指定模板参数 ch 为某个字符型常数
```

```
char fn()
{
    return ch-'A'+'a'; // 这里 ch 是模板参数
}
```

使用时有如下结果：

```
fn<'A'>(); // 将 ch 代换为 'A' 实现一个函数，返回 'a'
fn<'B'>(); // 将 ch 代换为 'B' 实现一个函数，返回 'b'
```

注意，这里模板参数与函数参数的区别在于，每更换一个模板参数，编译器都要重新“写”一个函数，因此模板参数的值必须是编译时就可以确定的。

与函数参数一样，模板参数也可以同时有多个。

19.8.2 模板类

将模板的概念应用于类，就有了模板类，举例如下：

```
template <class T> // 声明模板，T 代表某种类型
class my_class {
public:
    T *me;
    my_class(T t);
    ~my_class();
};
```

这时，类的各个成员函数必须也以模板的方式实现，如：

```
template <class T>
my_class<T>::my_class(T t) { me = new T(t); }
template <class T>
my_class<T>::~~my_class() { delete me; }
```

模板的声明只对后面的一个类或函数有效，因此每次定义一个新的模板函数或模板类，都要重新使用一次关键字 **template**。

一般来说，模板类和模板函数的整个实现都会放在头文件中，这样引用者才能知道所有细节以最终“写”出新的类或函数。这时，它有可能在多个源文件中被引用，但是编译器允许有这样的情况，不认为是重复定义。

19.9 异常处理

在 C 语言中，如果一个函数在执行中发现有错误，通常会返回一个特殊的值，调用者可以检测这个值并做相应的处理。这样，程序正常执行的代码与出错处理的代码是混杂在一起的，书写比较繁琐且不利于阅读。为了解决这个问题，C++ 语言中提出了新的错误处理机制，称为异常。异常处理有两个方面：抛出异常和捕捉异常。

当代码在执行过程中检测到错误时，可以用 **throw** 关键字抛出异常，如：

```
void fun(bool b) throw(int, const char *) // 声明函数可以抛出的异常类型
{
    if (b) {
        throw "Error!"; // 抛出 const char * 型的异常
    } else {
        throw 100; // 抛出 int 型的异常
    }
}
```

声明函数时可以同时声明它能抛出的异常的类型，如果不声明，则默认可以抛出任何类型的异常。如果函数不抛出异常，则可以写一个空的声明，如：

```
void fun(void) throw(); // 声明函数不抛出异常
```

异常可以是任何类型，包括各种类。抛出的异常是一个临时对象，这个对象在异常被处理之后才会销毁。一般都会将异常包装成各种类。例如，默认情况下，**new** 操作符在内存分配失败时会抛出 **bad_alloc** 类的异常。

捕获异常的方式是将可能产生异常的代码放在 **try catch** 语句块中，比如：

```
try {
    fun(true); // 可能抛出异常的代码
} catch (int i) { // 捕获 int 型异常
    // 在这里处理整型的异常
} catch (const char *s) { // 捕获 const char * 型异常
    // 在这里处理字符串型的异常
} catch (...) { // 捕获任何类型的异常
    // 在这里处理其他类型的异常
}
```

try 语句块中是可能抛出异常的代码，而其后的各个 **catch** 子句则用来捕捉异常。异常的捕捉是通过类型来匹配的，如果某个 **catch** 子句上的参数类型与抛出的异常相匹配，称为异常被捕捉，这个 **catch** 语句块中的代码将被执行，然后执行流程转移到所有 **catch** 块之后。

异常的处理过程可描述如下。

- ◆ 当代码执行到抛出异常时，如果代码在 **try** 语句块中，则执行流程转移到异常被捕捉的地方。
- ◆ 如果异常没有被捕捉，或者抛出异常的代码根本不在 **try** 语句块中，则当前函数将以异常状态返回，异常向它的调用者传播，等价于在调用者中发生了异常。
- ◆ 如果异常传播到函数的调用者后处在 **try** 语句块中，则执行流程转移到异常被捕捉的地方；如果异常仍然没有被捕捉，则继续向上逐级传播。
- ◆ 如果异常传播到最上层的函数仍然没有被捕捉，则系统的默认处理是以错误状态终止程序。
- ◆ 异常被处理之后，执行流程转移到所在 **try** 语句块对应的所有 **catch** 语句块之后。

有了异常处理机制后，代码中发现错误时只需抛出一个异常，对错误的处理则集中到了几个 **catch** 语句中，代码的可读性得到了提高，并且有可能降低代码量。

需要指出的是，异常的处理需要编译器生成一些额外的代码，既增加目标代码的体积，又可能降低程序执行的速度。这在嵌入式设备的软件开发中有时是不可容忍的，因此会在编译时禁止对异

常进行处理，如：

```
g++ -fno-exceptions sample.cpp # -fno-exceptions 参数禁止处理异常
```

这样源代码中就不能再使用异常处理机制。

19.10 C 与 C++ 混合编程

如果需要将已有的 C 语言代码模块与 C++ 语言代码模块相互链接，那么最快捷的办法就是直接将 C 语言代码当做 C++ 语言来编译。但在很多情况下还是不得不混合使用 C 语言和 C++ 语言进行编程。由于 C 语言和 C++ 语言在函数的命名和调用方式上均有不同，如果想在 C++ 源码中调用 C 语言模块中的函数，则需要做特殊的声明，例如：

```
extern "C" int fun(); // 声明 fun 为外部的 C 函数
```

这样，编译器就知道这个函数必须按 C 语言的方式调用，从而为其生成相应的目标代码。如果有多个函数，可以放在一起声明，如：

```
extern "C" {
    int fun1(void);
    void fun2(int, int);
}
```

通常，作为模块接口函数的声明都放在头文件中，以供多个源文件引用。如果一个头文件既被 C++ 源文件引用又被 C 源文件引用，则上述声明就出现了问题，因为 C 编译器不认识 `extern` 后的字符串 C。解决这个问题的方法是在头文件中加入以下代码，使之对 C 和 C++ 编译器通用：

```
#ifdef __cplusplus
extern "C" { // C 编译器看不到这一行
#endif
    int fun1(void);
    void fun2(int, int);
#ifdef __cplusplus
} // C 编译器看不到这一行
#endif
```

这里用到了一个宏定义 `__cplusplus`，如果通过 C 编译器进行预处理，则它没有定义；如果通过 C++ 编译器进行预处理，则它有定义。这样就可以把代码分为两种情况进行编译，从而实现对 C 和 C++ 编译器的通用性。



第 20 章 嵌入式 GUI 编程

嵌入式系统作为资源受限的系统，其显示设备也往往带有这一特征。用于嵌入式系统的显示设备往往尺寸较小、显示能力较弱，硬件的图形处理能力较弱或者根本没有专门的图形处理硬件。从应用上来说，嵌入式系统往往只显示专用应用程序的界面，不需要通用计算机那种复杂的桌面环境。

Qt 作为一种跨平台的图形界面开发平台，可以直接建立在简单的帧缓冲驱动上，并且有良好的可配置、可裁剪特性，因此也经常用在嵌入式系统上。Qt 支持一般图形界面系统的由事件驱动的编程模型，并且支持特有的信号与槽的编程模型，应用非常灵活。经过多年的发展，Qt 已经不单纯是一个图形界面开发平台，它对应用编程的各种领域（如网络、数据库等）都提供了自己的支持，功能非常强大。

在这一章里，我们将以 Qt 4.5.2 的开源版本为例，介绍如何使用 Qt 平台进行嵌入式 GUI 的开发。

20.1 建立开发环境

为了建立 Qt 4.5.2 的开发环境，需要从 Qt 的官方网站下载一些源码和软件。首先是 Qt 4.5.2 嵌入式版本的源码，可用以下方法下载：

```
wget http://get.qtsoftware.com/qt/source/qt-embedded-linux-opensource-src-4.5.2.tar.gz
```

这个源码包含了用于嵌入式开发的 Qt 4.5.2 共享库。

其次是 Qt 4.5.2 X11 版本的源码，可用以下方法下载：

```
wget http://get.qtsoftware.com/qt/source/qt-x11-opensource-src-4.5.2.tar.gz
```

这个源码主要用于得到一些 Qt 开发的工具，如 designer, qvfb 等。

安装的过程可分为三个部分：首先安装 Qt 的 X11 版本，然后对 Qt 的嵌入式版本进行两次编译，一次编译为本地 x86 版本，另一次编译为 ARM 目标机版本。

一、安装 Qt X11

下面介绍 Qt 4.5.2 X11 版本的安装过程。这里用的编译器是随 Debian 5.0 系统发行的 gcc-4.3.2 编译工具链。首先解压缩源码：

```
tar -xzf qt-x11-opensource-src-4.5.2.tar.gz
```

然后进入源码目录并进行配置：

```
cd qt-x11-opensource-src-4.5.2
```

```
./configure
```

配置时首先询问所用的 Qt 版本是商用的还是开源的，由于我们用的是开源版本，因此输入字母 **o**；然后询问是否接受 GPL 版权，输入 **yes** 即可继续配置过程。配置完成后就可以进行编译了：

```
make
```

编译完成后可用以下命令进行安装：

```
sudo make install
```

默认的安装目录是 `/usr/local/Trolltech/Qt-4.5.2`，因此需要 **root** 权限才能安装。如果想改变安装目录，可在配置时用 `-prefix` 选项指定，比如：

```
./configure -prefix ~/software/Qt-4.5.2
```

安装完毕以后还需要单独编译并安装嵌入式 Qt 开发的常用工具 **qvfb**，仍然在 Qt 4.5.2 X11 的源码目录中，输入以下命令进行编译：

```
make -C tools/qvfb/
```

编译完毕后得到的可执行文件在 **bin** 目录中，将其复制到安装目录中：

```
sudo cp bin/qvfb /usr/local/Trolltech/Qt-4.5.2/bin
```

将编译得到的 **designer**，**qvfb** 等工具链接到系统可执行文件目录以方便使用：

```
sudo ln -fsv /usr/local/Trolltech/Qt-4.5.2/bin/designer /usr/local/bin
sudo ln -fsv /usr/local/Trolltech/Qt-4.5.2/bin/qvfb /usr/local/bin
sudo ln -fsv /usr/local/Trolltech/Qt-4.5.2/bin/lupdate /usr/local/bin
sudo ln -fsv /usr/local/Trolltech/Qt-4.5.2/bin/lrelease /usr/local/bin
sudo ln -fsv /usr/local/Trolltech/Qt-4.5.2/bin/linguist /usr/local/bin
sudo ln -fsv /usr/local/Trolltech/Qt-4.5.2/bin/rcc /usr/local/bin
```

二、本地编译嵌入式 Qt

本地编译嵌入式 Qt 的目的是为了在本地编译嵌入式 Qt 应用程序，这样就可以利用 **qvfb** 在主机上进行调试了。这里用的编译器也是随 Debian 5.0 系统发行的 **gcc-4.3.2** 编译工具链。首先解压缩源码：

```
tar -xzf qt-embedded-linux-opensource-src-4.5.2.tar.gz
```

进入源码目录并进行配置：

```
./configure -no-largefile -no-accessibility -no-qt3support -no-phonon -no-svg
-no-nis -no-cups -no-opengl -qvfb
```

这里为了使编译出的共享库尽可能小，关闭了 Qt 的很多功能。配置的过程与安装 Qt X11 时相同。配置完成后就可以开始编译了：

```
make
```



编译完成后进行安装：

```
sudo make install
```

默认的安装目录是 `/usr/local/Trolltech/QtEmbedded-4.5.2`。

最后一步是设置环境变量，将 Qt 的可执行文件路径加入到 `PATH` 环境变量中，如：

```
export PATH=${PATH}:/usr/local/Trolltech/QtEmbedded-4.5.2/bin
```

三、交叉编译嵌入式 Qt

为了在 ARM 目标机上运行 Qt 应用程序，还必须交叉编译 Qt 的嵌入式版本。因为源码在编译本地版本时已经解压缩，所以可以直接进入源码目录进行配置，但配置前先要清空旧的配置，用以下命令：

```
make confclean
```

然后重新进行配置：

```
./configure -no-largefile -no-accessibility -no-qt3support -no-phonon -no-svg  
-no-nis -no-cups -no-opengl -prefix ${SYSROOT}/usr/local/Trolltech/  
QtEmbedded-4.5.2 -no-rpath -embedded arm -xplatform qws/linux-arm-g++
```

与本地编译时的配置相比，首先这里去掉了一个参数 `-qvfb`，因为目标机上的程序不需要在 `qvfb` 中运行。其次是增加了安装目录的配置以及交叉编译的配置。增加的参数 `-no-rpath` 表示编译时不要在可执行文件中记住所链接的共享库的路径，因为在目标机上运行时，共享库的路径与本地路径是不同的，记住也没有用。

其中引用的环境变量 `SYSROOT` 代表主机上的一个根文件系统，这里将 Qt 直接装在根文件中以便调试时使用。

配置完成后开始编译，这里用的编译工具链是 `arm-linux-gcc-3.4.4`：

```
make
```

编译完成后安装：

```
sudo make install
```

由于主机上现在同时存在嵌入式 Qt 的本地版本和 ARM 版本，使用时要注意 `PATH` 环境变量的设置。如果要编译 ARM 平台的应用程序，则应做如下设置：

```
export PATH=${PATH}:${SYSROOT}/usr/local/Trolltech/QtEmbedded-4.5.2/bin
```

20.2 简单的 Qt 应用程序

这一节里将通过一个简单的 Qt 应用程序来说明 Qt 应用编程的基本流程及编译和运行过程。



20.2.1 编写源代码

下面是一个简单的 Qt 应用程序的源代码：

```
// 文件名: main.cpp
// 说明: 简单 Qt 例程

#include <QApplication> // 使用 QApplication 类
#include <QWidget> // 使用 QWidget 类

int main(int argc, char *argv[])
{
    QApplication app(argc, argv); // 构造一个 QApplication 对象
    QWidget wdg; // 构造一个 QWidget 对象
    wdg.setWindowTitle("Widget"); // 设置窗口标题
    wdg.show(); // 让窗口成为显示状态
    return app.exec(); // 启动主事件循环
}
```

在这个源代码中使用了两个 Qt 提供的类 `QApplication` 和 `QWidget`。为了使用这两个类，必须包含它们的头文件。一般来说，每个 Qt 类都对应着一个头文件，这个头文件被包装成与类同名的一个文件以方便记忆，注意文件名上没有 `.h` 后缀。

`QApplication` 类代表应用程序，在每个程序中只能生成一个它的实例。在有图形界面的程序中必须有一个 `QApplication` 类的实例，并且必须在所有窗口类的实例生成之前生成。它最主要的功能是实现主事件循环。在主事件循环内，窗口才可以响应消息，并对事件做出处理。`QApplication` 类的 `exec` 方法代表启动主事件循环，这个函数在正常运行时不会返回，只有当主事件循环退出时才返回，通常这也就意味着整个程序要退出。

`QApplication` 类的构造函数接受与 `main` 函数相同的参数，它也能够处理执行程序时的命令行参数。这里的参数传递是必需的，因为在执行程序时要用到 `QApplication` 类才能识别的 `-qws` 参数。



可以对 `main` 函数传入的参数进行处理，甚至直接构造一个全新的参数列表，然后再传递给 `QApplication` 类的构造函数。

`QWidget` 类代表窗口，新生成的窗口默认是隐藏的，必须调用它的 `show` 方法才能使之成为显示状态。注意，窗口并不是在 `show` 方法调用完成后就真正显示在屏幕上，实际上只是发送了一个要求显示的事件，必须等主事件循环处理这个事件以后才真正显示出来。

在这个例程内，我们先生成了一个 `QApplication` 类的实例，然后生成一个 `QWidget` 类的实例，接着让 `QWidget` 的实例成为显示状态，最后启动主事件循环，这样一个窗口就会出现在屏幕上。这也是 Qt 程序的 `main` 函数的基本书写模式。

20.2.2 编译

写好的 Qt 应用程序在编译时要使用 Qt 的头文件，并且与 Qt 的各种共享库链接。对此，



Qt 4.5.2 提供了一个工具 **qmake**，可以省去手工写 **Makefile** 的麻烦。

使用 **qmake** 进行编译的第一步是生成工程文件，在应用程序的源码目录中执行以下命令：

```
qmake -project
```

生成的工程文件是源码目录的名称加上后缀 **.pro**，其内容如下：

```
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .
# Input
SOURCES += main.cpp
```

实际上，**qmake** 会搜索当前目录树下所有的 **.cpp**，**.c**，**.h** 等文件并把它们加入到工程文件中。这个文件也可以手动修改。

有了工程文件以后，就可以根据它生成 **Makefile** 了，执行以下命令：

```
qmake
```

这个命令将根据当前目录下的工程文件及源文件的内容生成一个 **Makefile**，然后就可以直接进行编译了：

```
make
```

默认情况下，编译完成后生成的可执行文件名称就是工程文件名去掉 **.pro** 后缀。

20.2.3 工程文件

有时候需要手动修改工程文件，因此有必要理解其中一些常用变量的含义，如表 20.1 所示。

表 20.1 常用工程文件变量

变量名	含义
TEMPLATE	模板，app 代表应用程序，lib 代表共享库
CONFIG	配置选项
QT	当配置选项中有 qt 时，表示需要链接的 Qt 库的列表
HEADERS	头文件列表
SOURCES	源文件列表
FORMS	ui 文件列表，由 designer 生成的界面文件
RESOURCES	资源文件列表
TARGET	目标的名称，默认与工程文件同名（去掉 .pro 后缀）
DESTDIR	目标放置的目录
DEFINES	编译时增加的宏定义列表
INCLUDEPATH	额外的头文件搜索路径
LIBS	额外链接的共享库，注意这里要写全给编译器的参数 -L，-l 等
VERSION	当模板是 lib 时，代表共享库的版本号



其中 CONFIG 变量有如下的常用值。

- ◆ qt: 表示工程为 Qt 应用程序, 这是默认值。
- ◆ release: 编译为 release 版本。
- ◆ debug: 编译为 debug 版本。
- ◆ debug_and_release: 同时编译为 release 和 debug 版本。
- ◆ warn_on: 编译时产生尽可能多的警告。
- ◆ warn_off: 编译时产生尽可能少的警告。
- ◆ thread: 表示工程为多线程应用程序。

当配置为 Qt 应用程序时, 程序要与 Qt 的共享库链接。Qt 4.5.2 提供了多个共享库, 分别支持不同的功能。可以用 QT 这个环境变量来控制所链接的共享库, 环境变量的值与共享库的对应关系请参考表 20.2。

表 20.2 QT 变量的值与共享库

QT 变量的值	对应的共享库	说明
core	QtCore	核心模块, 默认包含
gui	QtGui	图形界面模块, 默认包含
network	QtNetwork	网络模块
opengl	QtOpenGL	OpenGL 模块
sql	QtSql	SQL 模块
svg	QtSvg	SVG 模块
xml	QtXml	XML 模块
xmlpatterns	QtXmlPatterns	XML 模式模块
qt3support	Qt3Support	Qt3 支持模块

变量用 = 赋值, 则新值完全取代旧值; 如果用 += 赋值, 表示原值保留, 再增加一个新值; 如果用 -= 赋值, 则表示从原值中去掉一个值。例如:

```
CONFIG += warn_on # 增加配置选项
QT -= gui # 去掉 gui 选项, 非图形界面程序可以不链接 QtGui
```

20.2.4 运行

嵌入式版本的 Qt 应用程序通常以帧缓冲设备 /dev/fb0 作为显示设备。在主机上一般没有这个设备, 或者虽然有但是与主机的图形系统相冲突, 因此常用 Qt 的虚拟帧缓冲设备 qvfb 进行调试。用以下命令启动 qvfb:

```
qvfb -width 480 -height 272 &
```

这里在启动 qvfb 的同时指定了屏幕的宽度为 480 像素, 高度为 272 像素, 当然也可以在 qvfb 启动之后通过它的菜单修改大小。最后的符号 & 表示以后台方式执行这条命令, 因为接下来还要启动应用程序:

```
./app -qws # app 是应用程序的名称
```

这里的 `-qws` 使得程序以窗口服务器的方式启动。窗口服务器对帧缓冲设备和指定的输入设备进行独占式的管理，其他 Qt 图形界面程序通过与窗口服务器进行通信的方式来间接地使用这些设备。因此，第一个 Qt 图形界面程序必须以窗口服务器的方式启动，再启动其他 Qt 图形界面程序就不能是窗口服务器的方式了。

程序执行的结果如图 20.1 所示。

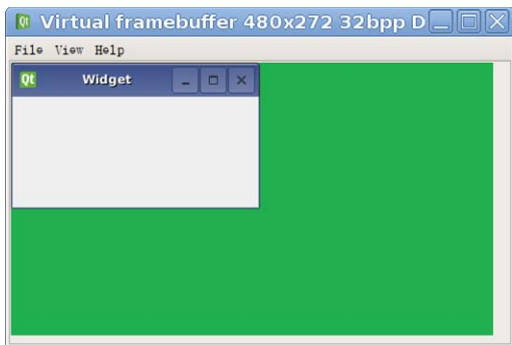


图 20.1 qvfb 中的窗口

20.2.5 移植到目标机

以上介绍的是在主机上使用 `qvfb` 进行模拟运行。为了在目标机上运行 Qt 应用程序，需要在目标机上建立 Qt 的运行环境，这需要以下几个步骤。

一、基本系统

首先确认已有的目标机根文件系统能够正常启动，并且帧缓冲驱动工作正常，即通过设备文件 `/dev/fb0` 能够在屏幕上输出图像。

二、复制 Qt 共享库

Qt 4.5.2 的共享库放在安装目录的 `lib` 子目录下。嵌入式平台由于资源有限，可以只安装其中一部分共享库。一般来说，一个图形界面程序至少依赖于 `QtCore` 库和 `QtGui` 库，因此最少要向目标机根文件系统中复制这两个库。可执行程序依赖的共享库可用以下命令查询：

```
arm-linux-readelf -d app # 查询 app 程序依赖的所有共享库
```

另外 Qt 共享库所依赖的 C++ 和 C 共享库也必须存在于根文件系统中，如 `stdc++` 库、`gcc_s` 库等。

三、复制字体

Qt 4.5.2 的字体放在安装目录的 `lib/fonts` 目录下，将需要的字体复制到目标机根文件系统中即可。

四、设置环境变量

由于目标机上 Qt 共享库所在的目录一般不会是主机上编译 Qt 时的共享库安装目录，所以需要在目标机上设置环境变量以通知 Qt 应用程序共享库和字体所在的目录，如：

```
export LD_LIBRARY_PATH=/usr/local/Trolltech/QtEmbedded-4.5.2/lib:${LD_LIBRARY_PATH}
export QT_QWS_FONTPATH=/usr/local/Trolltech/QtEmbedded-4.5.2/lib/fonts
```

环境变量 `LD_LIBRARY_PATH` 用于控制程序运行时共享库的搜索路径，所以要将 Qt 共享库所在的路径加进去。环境变量 `QT_QWS_FONTPATH` 则用于指定字体的存放路径。一般会把这些命令加入到目标机的 Shell 登录脚本中，使之每次重启后都生效。

设置好目标机的运行环境后，在主机上重新设置 Qt 的编译环境，编译为目标机版本的可执行程序，然后将它复制到目标机根文件系统中去。第一个启动的 Qt 程序仍然需要是窗口服务器，如：

```
/app -qws # 以窗口服务器方式启动程序 /app
```

20.3 窗口布局

布局是 Qt 中用来管理窗口内的子窗口的一种方法。布局本身不是窗口，它类似于一个虚拟的看不见的容器，可以自动调整其内的子窗口或子布局的大小和位置。

布局统一由 `QLayout` 类代表，以它作为基类，派生出所有具体的布局类型。其中常用的布局有水平布局、垂直布局和栅格布局，它们分别由 `QHBoxLayout` 类，`QVBoxLayout` 类和 `QGridLayout` 类代表。这些类的继承关系如图 20.2 所示。

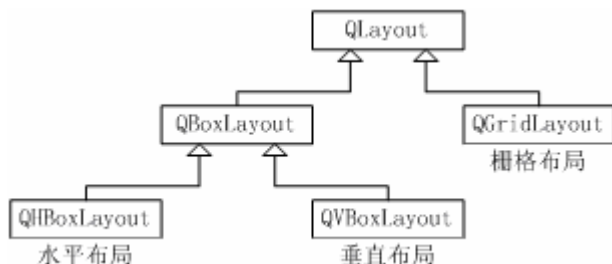


图 20.2 常用布局类的继承关系

20.3.1 水平布局与垂直布局

水平布局与垂直布局是两种最基本的布局形式，其内的各个子窗口按照水平或垂直的方式排列。这里用一个例程来说明它们的使用。例程共有三个源文件：`main.cpp`，`mywidget.h` 和 `mywidget.cpp`。其中 `main.cpp` 的源码如下：

```
// 文件名: main.cpp
// 说明: 水平布局和垂直布局例程主模块

#include <QApplication> // 使用 QApplication 类

#include "mywidget.h" // 使用 MyWidget 类

int main(int argc, char *argv[])
{
    QApplication app(argc, argv); // 构造一个 QApplication 对象
```



```
MyWidget wdg; // 构造一个 MyWidget 对象
wdg.show(); // 让窗口成为显示状态
return app.exec(); // 启动主事件循环
}
```

主模块中仍然是先生成 `QApplication` 对象，后生成窗口，然后显示窗口并启动主事件循环的模式，只不过窗口换成了自定义的窗口类 `MyWidget`。这个类的定义在 `mywidget.h` 头文件中，其源码如下：

```
// 文件名: mywidget.h
// 说明: 水平布局和垂直布局例程 MyWidget 类定义

#ifndef MYWIDGET_INCLUDED
#define MYWIDGET_INCLUDED

#include <QWidget> // 使用 QWidget 类

// 以下为类的前向声明
class QHBoxLayout;
class QVBoxLayout;
class QPushButton;
class QLabel;

class MyWidget: public QWidget { // MyWidget 继承 QWidget
public:
    MyWidget(); // 构造函数
    virtual ~MyWidget(); // 析构函数
protected:
    QHBoxLayout *layoutH; // 水平布局
    QVBoxLayout *layoutV; // 垂直布局
    QPushButton *button; // 按钮
    QLabel *label1, *label2; // 两个标签
};

#endif
```

这个类里定义了几个保护权限的成员变量，这些变量都是指向布局或者子窗口的指针。这里 `QPushButton` 类代表按钮，`QLabel` 类代表标签，它们都是 `QWidget` 类的子类，因而都属于窗口类，但通常这些类不作为独立窗口出现，而是依附于其他窗口之上，因此常称它们为窗口部件。为了定义这些类的指针，必须对它们进行类的前向声明。当然，直接包含这些类的头文件也是可以的，但那样会造成预处理以后的文件过大。编程的原则是只在必要的时候才包含头文件。

`MyWidget` 类的实现在源文件 `mywidget.cpp` 中，其源码如下：

```
// 文件名: mywidget.cpp
// 说明: 水平布局和垂直布局例程 MyWidget 类实现

#include "mywidget.h"

#include <QHBoxLayout>
```



```

#include <QVBoxLayout>
#include <QPushButton>
#include <QLabel>

MyWidget::MyWidget()
{
    setWindowTitle("Layout"); // 设置窗口标题
    layoutH = new QHBoxLayout(this); // 生成水平布局
    button = new QPushButton("PushButton"); // 生成按钮
    layoutH->addWidget(button); // 将按钮放入水平布局
    layoutV = new QVBoxLayout(); // 生成垂直布局
    layoutH->addLayout(layoutV); // 将垂直布局放入水平布局
    label1 = new QLabel("Label One"); // 生成标签 1
    layoutV->addWidget(label1); // 将标签 1 放入垂直布局
    label2 = new QLabel("Label Two"); // 生成标签 2
    layoutV->addWidget(label2); // 将标签 2 放入垂直布局
}

MyWidget::~MyWidget()
{
}

```

在这里，窗口内的所有部件都在构造函数内生成。首先生成一个水平布局，然后生成一个按钮放入水平布局内，再生成一个垂直布局放入水平布局内。在垂直布局内又先后放入了两个标签。需要注意的是，布局并不是窗口类，因此布局内放入窗口部件和放入布局的函数是不同的，一个是 `addWidget`，一个是 `addLayout`。整个窗口内的所有布局和部件的包含关系如图 20.3 所示。

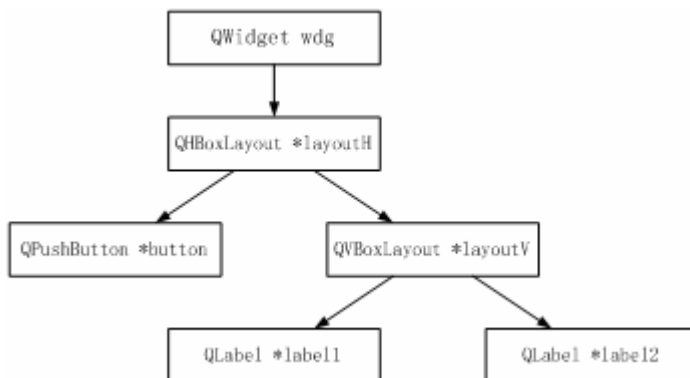


图 20.3 布局与部件的包含关系

对于一个窗口来说，只能有一个布局成为它的顶级布局，这可以通过向布局的构造函数传递 `QWidget` 类的指针来实现，也可以通过 `QWidget` 类的以下成员函数实现：

```
void setLayout(QLayout *layout);
```

这个函数可以将布局 `layout` 设为窗口的顶级布局。

如图 20.4 所示是例程的运行结果。可以看到，由于布局的作用，所有窗口部件以及窗口的大小都自动进行了调整。对于水平布局，先放入的项目在左；对于垂直布局，先放入的项目在上。

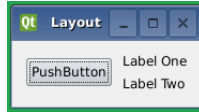


图 20.4 布局例程运行结果

20.3.2 栅格布局

栅格布局是另外一种常用的布局。这种布局将把窗口划分为大小相等的若干个区域，每个区域内可以放入一个窗口部件或子布局。这里用例程来说明它的使用。例程同样包含三个文件：`main.cpp`、`mywidget.h` 和 `mywidget.cpp`。其中 `main.cpp` 文件与水平布局和垂直布局例程中相同，`mywidget.h` 文件的内容修改如下：

```
// 文件名: mywidget.h
// 说明: 栅格布局例程 MyWidget 类定义

#ifndef MYWIDGET_INCLUDED
#define MYWIDGET_INCLUDED

#include <QWidget> // 使用 QWidget 类

// 以下为类的前向声明
class QGridLayout;
class QPushButton;
class QLabel;

class MyWidget: public QWidget { // MyWidget 继承 QWidget
public:
    MyWidget(); // 构造函数
    virtual ~MyWidget(); // 析构函数
protected:
    QGridLayout *layout; // 栅格布局
    QPushButton *button1, *button2; // 两个按钮
    QLabel *label1, *label2; // 两个标签
};

#endif
```

`mywidget.cpp` 文件的内容修改如下：

```
// 文件名: mywidget.cpp
// 说明: 栅格布局例程 MyWidget 类实现

#include "mywidget.h"

#include <QGridLayout>
#include <QPushButton>
#include <QLabel>

MyWidget::MyWidget()
```




```

{
    setWindowTitle("Layout"); // 设置窗口标题
    layout = new QGridLayout(this); // 生成栅格布局
    button1 = new QPushButton("PushButton One"); // 生成按钮 1
    button2 = new QPushButton("PushButton Two"); // 生成按钮 2
    label1 = new QLabel("Label One"); // 生成标签 1
    label2 = new QLabel("Label Two"); // 生成标签 2
    layout->addWidget(button1, 0, 0); // 将按钮 1 放入栅格布局 0 行 0 列
    layout->addWidget(label1, 0, 1); // 将标签 1 放入栅格布局 0 行 1 列
    layout->addWidget(label2, 1, 0); // 将标签 2 放入栅格布局 1 行 0 列
    layout->addWidget(button2, 1, 1); // 将按钮 2 放入栅格布局 1 行 1 列
}

MyWidget::~MyWidget()
{
}

```

在窗口的构造函数里，首先生成了一个栅格布局，然后将四个窗口部件放入这个布局，形成 2 行 2 列的排列。注意栅格布局的 `addWidget` 函数原型与水平布局和垂直布局是不同的，参数上需要指明行号和列号。当然，栅格布局内也可以嵌套其他布局，使用以下函数：

```
void addLayout(QLayout *layout, int row, int column);
```

其中参数 `layout` 指向子布局，`row` 为行号，`column` 为列号。

例程的运行结果如图 20.5 所示。



图 20.5 栅格布局例程运行结果

20.4 Qt 对象

Qt 中很多类都是由 `QObject` 类派生出来的，因此它们的实例都具有 `QObject` 类对象的特征，举例如下。

- ◆ 有一个字符串形式的名称。
- ◆ 支持信号与槽机制。
- ◆ 能够过滤和接收事件。
- ◆ 支持定时器，启动定时器后每隔一定的时间发生一次定时器事件。
- ◆ 实施层次化管理，即对象间有父子关系，父对象保存了所有子对象的指针，子对象保存了父对象的指针。
- ◆ 不能使用复制构造函数和赋值操作符。

本书中将这类统称为 Qt 对象类，将这些类的实例统称为 Qt 对象。为了支持这些特征，

一个 Qt 对象类不能只是简单地继承 QObject 类，还必须有一些特殊的写法，如：

```
class MyObject: public QObject {  
    Q_OBJECT  
    // 其他内容  
};
```

这里，类里面的第一行是一个宏 Q_OBJECT，随后是其他内容。如果只用 C++ 编译器去编译，则这个宏等于没写。实际上，Qt 有一个工具 moc 可以识别这个宏，它会为所有包含 Q_OBJECT 宏的类生成额外的源代码以支持 Qt 对象的所有特征。



如果一个类继承了 QObject 类，但没有使用任何 Qt 对象的特征，则可以不加 Q_OBJECT 宏。推荐在所有 QObject 的派生类中都加上 Q_OBJECT 宏。

当 qmake 生成 Makefile 时，会自动为需要 moc 工具处理的头文件生成相应的规则，注意它只对头文件进行处理，因此最好把类的定义放在头文件中。

Qt 对象还有属性的概念，通过属性可以获得对象的一些状态值。一个属性一般对应着一个与属性同名的只读成员函数，用来获得属性的值。如果属性是 bool 型，则对应的函数往往是 is 加上属性名。对于可写的属性，则还应有一个以 set 加上属性名为名称的成员函数用来设置属性的值。注意 Qt 的变量与函数的命名风格，一般遵循第一个单词全小写、随后的单词首字母大写的原则，因此上述函数名与属性名的大小写会有所不同。

20.4.1 层次化管理

在上一节的布局例程中，窗口内的布局和部件都定义为了一个成员指针，在窗口的构造函数里使用 new 操作符动态生成，但是在窗口的析构函数中并没有将这些对象 delete。这种用法是正确的，因为 Qt 对象有自己的内存管理策略，当父对象被析构时，会将它的所有子对象同时析构。因此，只要一个 Qt 对象的父对象设置适当，它就会在父对象析构时自动析构，不需要做多余的 delete 操作。



由于窗口内的部件会在窗口析构时自动用 delete 操作符析构，因此这些部件不能定义为成员变量，只能由 new 操作符动态生成。

各种 Qt 对象类的构造函数一般都可以接受一个 Qt 对象指针作为参数，用于设置父对象，这样在 Qt 对象生成时就确定了它的父对象。另外，有些函数可以改变 Qt 对象的父对象，如布局类的 addWidget 函数，它会使加入布局的窗口部件成为布局所在窗口的子对象。以上一节的水平布局和垂直布局例程为例，窗口与其内的所有布局和部件的父子关系如图 20.6 所示。

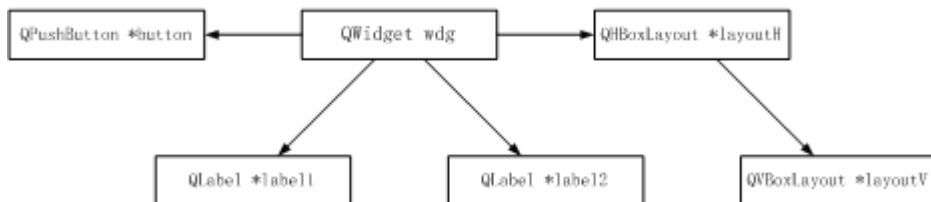


图 20.6 窗口与其内的所有布局和部件的父子关系

这里的父子关系显然与图 20.3 所示的包含关系不同。窗口内的布局和部件是分开管理的，只有最顶级的布局的父对象是窗口，其他布局的父对象都是包含它的布局，而窗口部件不管在哪个布局内，其父对象都是窗口本身。

20.4.2 信号与槽

信号与槽是一种 Qt 特有的对象间通信的机制，是 Qt 区别于其他图形系统的基本特征。信号与槽之间是一种松散的连接关系，这种连接关系可以在程序运行时动态改变，这为 Qt 编程提供了极大的灵活性。这里将通过一个例程说明信号与槽的使用，例程包括三个文件：main.cpp，myobject.h 和 myobject.cpp。

只有从 QObject 类派生出来的类才能使用信号与槽，因此首先要定义一个这样的类，其源码如下：

```
// 文件名: myobject.h
// 说明: 信号与槽例程 MyObject 类定义

#ifndef MYOBJECT_INCLUDED
#define MYOBJECT_INCLUDED

#include <QObject>

class MyObject: public QObject { // 必须继承 QObject
    Q_OBJECT // 使用信号与槽必须有
public:
    MyObject(): m_value(0) { } // 构造函数
protected:
    int m_value;
signals: // 信号
    void valueChanged(int value);
public slots: // 槽
    int setValue(int value);
};

#endif
```

定义信号使用 Qt 的 signals 关键字。注意信号没有访问权限，实际上它们都是保护权限的，也就是说只有定义信号的类自己及派生类才能发射这个信号。信号一般定义为无返回值的函数，这个函数不需要实现（由 moc 工具自动生成的代码实现）。

定义槽使用 Qt 的 slots 关键字。槽同时也是一个普通函数，可以有各种访问权限。文件 myobject.cpp 内实现了 MyObject 类中定义的槽，源码如下：

```
// 文件名: myobject.cpp
// 说明: 信号与槽例程 MyObject 类实现

#include <QtDebug> // 使用 qDebug()
#include "myobject.h"
```



```
int MyObject::setValue(int value)
{
    if (value != m_value) { // 判断值是否变化以免无限循环
        m_value = value;
        qDebug() << objectName() << ":" << "emit signal" << m_value;
        emit valueChanged(m_value); // 发射信号
        qDebug() << objectName() << ":" << "signal emitted";
    } else {
        qDebug() << objectName() << ":" << "value not changed";
    }
    return m_value;
}
```

这里使用了 `qDebug()` 函数来输出调试信息以便观察结果，它返回一个 `QDebug` 类的对象，可以支持用左移位操作符 `<<` 输出信息，默认情况下输出信息出现在标准错误输出上。`objectName()` 函数则是 `QObject` 类的一个成员，用于得到对象的名称。

发射信号使用 Qt 的 `emit` 关键字。信号发射后，与这个信号连接的槽就会被调用。在主模块 `main.cpp` 中使用了 `MyObject` 类的信号和槽，源码如下：

```
// 文件名: main.cpp
// 说明: 信号与槽例程主模块

#include "myobject.h"

int main(void)
{
    MyObject m, n; // 生成两个 MyObject 对象
    m.setObjectName("MyObject m"); // 设置对象 m 的名称以便在调试信息中使用
    n.setObjectName("MyObject n"); // 设置对象 n 的名称以便在调试信息中使用
    // 将对象 m 的 valueChanged 信号连接到对象 n 的 setValue 槽
    QObject::connect(&m, SIGNAL(valueChanged(int)), &n, SLOT(setValue(int)));
    // 将对象 n 的 valueChanged 信号连接到对象 m 的 setValue 槽
    QObject::connect(&n, SIGNAL(valueChanged(int)), &m, SLOT(setValue(int)));
    // 直接调用对象 m 的槽
    m.setValue(10);
    return 0;
}
```

在主函数中，首先生成了两个 `MyObject` 类的对象并为每个对象设置了名称以便在调试信息中区分，然后将两个对象的信号与槽进行交叉连接。连接信号与槽可以使用 `QObject` 类的以下静态成员函数：

```
bool connect(const QObject *sender, const char *signal,
            const QObject *receiver, const char *method,
            Qt::ConnectionType type = Qt::AutoConnection);
```

其各个参数及返回值的含义解释如下。

◆ `sender`: 指向 Qt 对象，信号的发送者。



- ◆ **signal**: 表示信号的字符串。
- ◆ **receiver**: 指向 Qt 对象, 信号的接收者。
- ◆ **method**: 表示与信号连接的方法的字符串, 这里的方法可以是槽或信号。
- ◆ **type**: 表示连接方式。
- ◆ **返回值**: **true** 表示连接成功, **false** 表示连接失败。

信号与槽的连接方式有以下几种。

- ◆ **Qt::DirectConnection**: 直接方式, 信号发射时将直接调用槽。
- ◆ **Qt::QueuedConnection**: 队列方式, 信号发射时产生一个事件进入队列, 事件被处理时槽才被调用。
- ◆ **Qt::BlockingQueuedConnection**: 阻塞队列方式, 信号发射时产生一个事件进入队列, 然后当前线程进入等待状态, 直到事件处理完毕, 仅用于多线程的情况。
- ◆ **Qt::AutoConnection**: 自动方式, 由系统自动选择连接方式。

一般来说都采用默认自动方式进行连接。因为支持信号与槽的类都是 **QObject** 的派生类, 因此如果在成员函数中进行连接可以不写 **QObject::** 这样的域限定符。

用来表示信号和槽的参数都是字符串, Qt 提供了两个宏用于构造这样的字符串: 对于信号使用 **SIGNAL**, 对于槽则使用 **SLOT**, 用它们将函数的原型包围起来即可。注意这里的函数原型只能写出类型, 不能有任何参数名, 否则连接时将会失败。

信号与槽的连接有以下特点。

- ◆ 一个信号可以连接到多个槽。
- ◆ 一个槽可以被多个信号连接。
- ◆ 信号也可以连接到信号, 此时前者的发射将导致后者的发射。
- ◆ 信号的参数类型必须与槽的参数类型对应, 信号的参数可以比槽的参数多, 但不可以少, 否则连接将失败。



定义信号与槽时最好不要有自定义类型的参数, 否则它只能与特定参数类型的信号或槽连接, 可用性大大降低。

已经建立的连接关系也可以被断开, 用 **QObject** 类的以下静态成员函数:

```
bool disconnect(const QObject *sender, const char *signal,
               const QObject *receiver, const char *method);
```

这里各个参数及返回值的含义与 **connect** 函数相同。需要注意的是, 除了参数 **sender**, 其他参数都可以是 0, 代表“所有”, 例如:

```
QObject::disconnect(myObject, 0, 0, 0);
```

这条语句将断开 **myObject** 对象的所有信号的所有连接。

例程的运行结果如下:

```
"MyObject m" : emit signal 10
"MyObject n" : emit signal 10
"MyObject m" : value not changed
"MyObject n" : signal emitted
"MyObject m" : signal emitted
```

从中可以看出,这里的信号与槽自动采用了直接连接的方式,因此在发射信号时,所连接的槽被调用,在槽函数执行完毕返回后,发射信号的后续语句才得以执行。

如果将 `main.cpp` 中的两个连接关系强行改为 `Qt::QueuedConnection` 方式,则发射信号时,槽不会被调用,因为没有一个是 `QApplication` 对象,事件得不到处理。在源码中再生成一个 `QApplication` 对象 `app`,并在最后调用 `app.exec()` 以后,得到的结果如下:

```
"MyObject m" : emit signal 10
"MyObject m" : signal emitted
"MyObject n" : emit signal 10
"MyObject n" : signal emitted
"MyObject m" : value not changed
```

从这里可以看出,信号发射以后,槽函数并没有被调用,程序就继续执行了。随后当事件得到处理以后,对应的槽函数才被调用。

需要指出的是,常用的 `Qt` 对象类中已经定义了很多有用的信号和槽,编程时可以直接使用。

20.4.3 事件

事件是 `Qt` 的另外一种通信机制。事件由两个要素组成,一是接收者,它必须为 `Qt` 对象;二是事件对象,保存了要传递给接收者的信息。事件有两种发送方式,一种为同步方式,在这种方式下,发送事件将直接进入事件处理流程,事件处理完毕之后,发送操作才成功返回,在这种情况下,发送者可以得到事件处理的结果;另一种为异步方式,在这种方式下,发送的事件会进入一个队列,这时发送操作就成功返回了,队列中的事件由主事件循环取出并进行处理,在这种情况下,发送者无法得到事件处理的结果。

对象处理事件时,如果安装了事件过滤器,则首先要由过滤器对事件进行筛选,没有被过滤掉的事件才进入对象自己的处理流程。

下面由一个例程来说明事件的发送、处理及过滤器的使用。例程同样由三个文件组成:`main.cpp`, `myobject.h` 和 `myobject.cpp`。首先是 `MyEvent` 事件类及 `MyObject` 类的定义,源码如下:

```
// 文件名: myobject.h
// 说明: 事件例程 MyEvent 类及 MyObject 类定义

#ifndef MYOBJECT_INCLUDED
#define MYOBJECT_INCLUDED

#include <QObject>
#include <QEvent>

// 自定义事件类 MyEvent, 它可以携带一个整数作为信息
class MyEvent: public QEvent { // 所有的事件类都由 QEvent 派生
```

```

public:
    // 构造时将 QEvent 的类型初始化为 User, 并保存整数值
    MyEvent(int i): QEvent(QEvent::User), m_value(i) { }
    int value() { return m_value; } // 返回所保存的整数值
protected:
    int m_value; // 用于保存一个整数值
};

class MyObject: public QObject { // 必须继承 QObject
    Q_OBJECT // 事件的相关处理并不需要这个宏, 建议加上
public:
    MyObject(); // 构造函数
    bool event(QEvent *e); // 覆盖事件处理函数
    bool eventFilter(QObject *obj, QEvent *e); // 覆盖事件过滤函数
};

#endif

```

MyEvent 类已在上述头文件中实现, 而 MyObject 类的实现则在下面的源文件中:

```

// 文件名: myobject.cpp
// 说明: 事件例程 MyObject 类实现

#include <QtDebug> // 使用 qDebug()
#include "myobject.h"

MyObject::MyObject()
{
    installEventFilter(this); // 安装事件过滤器
}

// 事件处理函数, 参数 e 指向接收到的事件
bool MyObject::event(QEvent *e)
{
    if (e->type() != QEvent::User) return false; // 无法处理类型非 User 的事件
    MyEvent *me = (MyEvent *)e; // 转换类型
    qDebug() << "MyEvent" << me->value() << "processed";
    return true; // 处理完毕, 返回 true
}

// 事件过滤函数, 参数 obj 指向接收者, 参数 e 指向事件
bool MyObject::eventFilter(QObject *obj, QEvent *e)
{
    // 对于无法过滤的事件, 交由父类过滤
    if (e->type() != QEvent::User) return QObject::eventFilter(obj, e);
    MyEvent *me = (MyEvent *)e; // 转换类型
    if (me->value() == 0) { // 判断, 只过滤值为 0 的事件
        qDebug() << "MyEvent" << me->value() << "dropped";
        return true; // 事件被过滤掉, 返回 true
    } else {
        qDebug() << "MyEvent" << me->value() << "pass";
    }
}

```



```

        return QObject::eventFilter(obj, e); // 不过滤的事件交由父类过滤
    }
}

```

这里在 `MyObject` 类的对象构造时为其安装了一个事件过滤器。事实上，每个 `Qt` 对象都是一个天然的事件过滤器，其过滤操作由成员函数 `eventFilter` 决定。这个函数接受两个参数，一个指向事件的接收者，一个指向事件对象，在函数内可以根据这两个参数进行不同的过滤操作。如果要把事件过滤掉，则返回 `true`；如果要让事件通过，则返回 `false`。这是一个虚函数，`QObject` 类的派生类可以覆盖它。一般来说，如果事件没有被过滤掉，则应交由父类进行过滤，以免遗漏重要的事件。

安装过滤器使得一个对象能够介入另一个对象的事件处理流程，极大地增加了编程的灵活性。一个对象还可以安装多个不同的事件过滤器，这时后安装的事件过滤器将先被调用。安装的过滤器既可以是自身，也可以是其他对象。常见的做法是父对象把自己作为过滤器安装给子对象，这样就使得多个子对象对事件有了相同的反应。

安装的过滤器还可以移除，用 `QObject` 类的以下成员函数：

```
void removeEventFilter(QObject *obj);
```

其中参数 `obj` 指向要移除的过滤器。

在主模块内，首先生成 `MyObject` 类的对象，然后向它发送事件，源码如下：

```

// 文件名: main.cpp
// 说明: 事件例程主模块

#include <QtDebug>
#include <QApplication>
#include "myobject.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv); // 生成 QApplication 对象
    MyObject m;
    qDebug() << "post MyEvent 0";
    app.postEvent(&m, new MyEvent(0)); // 异步发送事件 0
    qDebug() << "send MyEvent 0";
    app.sendEvent(&m, new MyEvent(0)); // 同步发送事件 0
    qDebug() << "post MyEvent 1";
    app.postEvent(&m, new MyEvent(1)); // 异步发送事件 1
    qDebug() << "send MyEvent 1";
    app.sendEvent(&m, new MyEvent(1)); // 同步发送事件 1
    return app.exec();
}

```

这里使用了 `QApplication` 类的以下两个成员函数来发送事件：

```

void postEvent(QObject *receiver, QEvent *event); // 异步发送事件
bool sendEvent(QObject *receiver, QEvent *event); // 同步发送事件

```



其中参数 `receiver` 指向接收者, 参数 `event` 指向要发送的事件。同步发送的返回值就是事件过滤或处理的返回值, 而异步发送没有返回值。

例程的运行结果如下:

```
post MyEvent 0
send MyEvent 0
MyEvent 0 dropped
post MyEvent 1
send MyEvent 1
MyEvent 1 pass
MyEvent 1 processed
MyEvent 0 dropped
MyEvent 1 pass
MyEvent 1 processed
```

从中可以看出, 异步发送的事件并没有立刻得到处理, 而同步发送时事件得到处理以后才返回。还可以看到所安装的事件过滤器发挥了作用。

这里的事件是在代码中通过函数调用发送的, 实际上, Qt 中定义的大多数事件都是系统发出的, 尤其是与窗口有关的输入输出事件。一般来说, Qt 提供的类会为每个所支持的事件定义一个形如 `xxxEvent(QxxxEvent *)` 的虚成员函数, 因此大多数情况下只需要覆盖具体的某个事件对应的函数, 而不需要重新实现 `event()` 函数。

对于异步发送的事件, 如果程序的执行流程没有回到主事件循环, 则它们不会被处理。可以调用 `QApplication` 类的以下成员函数使排队的事件得到处理:

```
void processEvents(QEventLoop::ProcessEventsFlags flags =
    QEventLoop::AllEvents);
```

其默认参数表示处理队列中的所有事件。

因为一个应用程序中只能有一个 `QApplication` 对象, 所以 Qt 定义了一个全局指针 `qApp`, 指向这个唯一的 `QApplication` 对象, 这样就可以随处访问这个对象了, 如:

```
qApp->processEvents(); // 处理队列中的事件
```

当然, 访问时必须包含头文件 `<QApplication>`。

20.4.4 定时器

20.4.4.1 定时器事件

定时器事件是每个 Qt 对象都支持的一种定时方式。在 `QObject` 类中, 可以使用下面的成员函数启动一个定时器:

```
int startTimer(int interval);
```

其中参数 `interval` 是定时器的周期, 单位是毫秒。函数的返回值是定时器的标识, 一个 Qt 对象中可以启动多个定时器, 这就需要用标识来区分它们。如果启动定时器失败, 则返回值为 0。定时器启动之后, 每过一个周期的时间, 就会发生一次定时器事件, 其对应的处理函数原型如下:



```
void timerEvent(QTimerEvent *event);
```

这是 `QObject` 类的一个保护权限虚成员函数，其中 `QTimerEvent` 类型表示定时器事件，它有一个成员函数可以得到发送事件的定时器的标识：

```
int timerId() const;
```

如果想关闭定时器则使用 `QObject` 类的以下成员函数：

```
void killTimer(int id);
```

其中参数 `id` 是要关闭的定时器的标识。

20.4.4.2 QTimer

Qt 平台另外提供了一个类 `QTimer` 专门用于定时操作。实际上，它是由 `QObject` 类派生出来的，并将 `QObject` 的定时器事件机制包装成了信号机制。

`QTimer` 类的构造函数原型如下：

```
QTimer(QObject *parent = 0);
```

与一般的 `QObject` 对象一样，`QTimer` 对象也可以有一个父对象，这样它就会在父对象析构时自动析构。

`QTimer` 类的以下成员函数可以设置定时器的周期：

```
void setInterval(int msec);
```

其中参数 `msec` 是要设置的周期，单位是毫秒。实际上 `interval` 是 `QTimer` 类的一个属性，因此有一个函数可以返回它的值：

```
int interval() const;
```

`QTimer` 还有一个 `bool` 型的属性 `singleShot`，它是一个可写属性。如果这个属性设置为 `true`，则超时信号只发生一次，而不是周期性地发生。与之相关的函数如下：

```
bool isSingleShot() const; // 读 singleShot 属性  
void setSingleShot(bool singleShot); // 写 singleShot 属性
```

当设定的时间超过之后，`QTimer` 对象就会发射超时信号，原型如下：

```
void timeout();
```

将这个信号与另外一个 Qt 对象的某个槽连接起来，就可实现对超时信号的处理。

`QTimer` 对象生成之后并不处在激活状态，可用以下成员函数启动和停止：

```
void start(int msec); // 启动定时器，同时设置周期  
void start(); // 启动定时器  
void stop(); // 停止定时器
```



20.5 使用 designer

应用程序的一个窗口上通常有很多部件,如果这些都需要自己写代码生成,则各个部件的大小、位置、颜色及相互间的关系等处理起来是一项非常繁琐的工作。Qt 4.5.2 提供了一个界面设计工具 **designer**, 能够以可视化的方式编辑窗口外观, 并通过 Qt 的 **uic** 工具转化为 C++ 源代码, 因此可以大大减少界面开发的工作量。

designer 工具在编译 Qt X11 版本时已产生, 在窗口环境下打开终端, 输入以下命令就可以启动:

```
designer &
```

这里仍然以后台执行的方式启动以免它占用终端。启动以后的界面如图 20.7 所示。

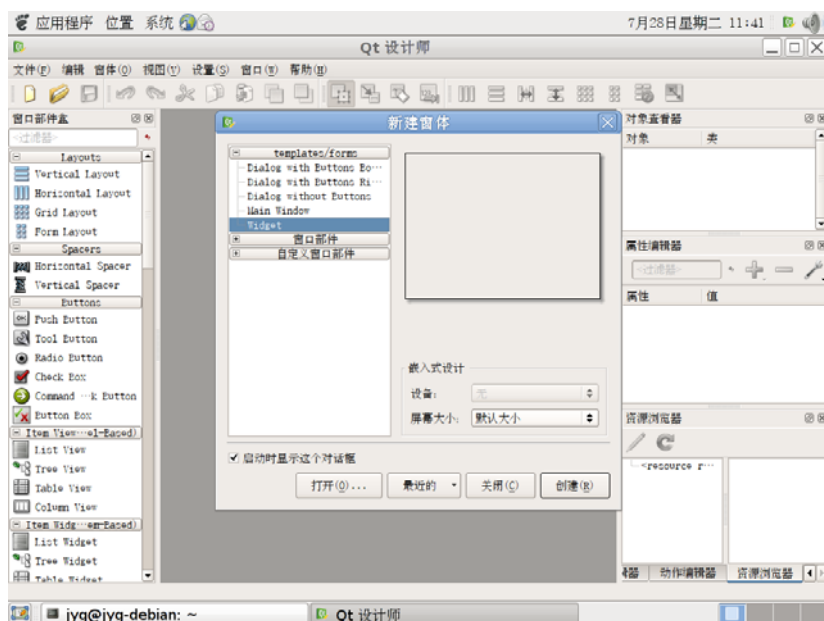


图 20.7 designer 界面

下面将通过一个例程简单说明 **designer** 的使用。在这个例程的窗口上将显示一个编辑框、一个按钮和一个标签, 用户可以向编辑框中输入一些文字, 然后单击按钮, 编辑框中的文字将被复制到标签上。

20.5.1 窗体设计

例程的界面设计按下面描述的步骤进行。

一、新建窗体

首先新建一个 **Widget** 形式的窗体, 它是基于 **QWidget** 类的窗体。然后在属性编辑器中修改它的 **objectName** 属性的值为 **MyWidget**, 这与将来生成源码时窗体对应的类名有关。再修改它的 **windowTitle** 属性为 **Copy String**, 这是窗体的标题。

二、添加部件

从窗口部件盒中拖动部件到窗体就可以向窗体添加部件。添加一个编辑框、一个按钮和一个标签，它们的默认名称是 `lineEdit`，`pushButton` 和 `label`，这也是将来用于访问这些窗口部件的变量名。将按钮的 `text` 属性修改为 `Copy`，这是按钮上要显示的文本。

三、设置布局

一般来说，窗体都应该有一个顶级布局，这样当窗体的大小变化时，所包含的部件也会自动调整。当窗体内有了部件以后就可以设置顶级布局了，方法是：在窗体上单击鼠标右键，在弹出的菜单中选择“布局”选项，其子菜单上是各种布局的方式，这里我们选择“垂直布局”选项，即可看到所有部件按照垂直方向排列好了。然后调整窗口到合适的大小。

四、添加槽

为了实现用户单击按钮时有所动作，必须在窗口内建立一个槽。方法是：在窗体上单击鼠标右键，在弹出的菜单中选择“改变信号/槽”选项，即可出现信号/槽编辑对话框，如图 20.8 所示，单击上方的加号按钮即可增加新的槽，这里增加了一个名为 `copy()` 的槽。

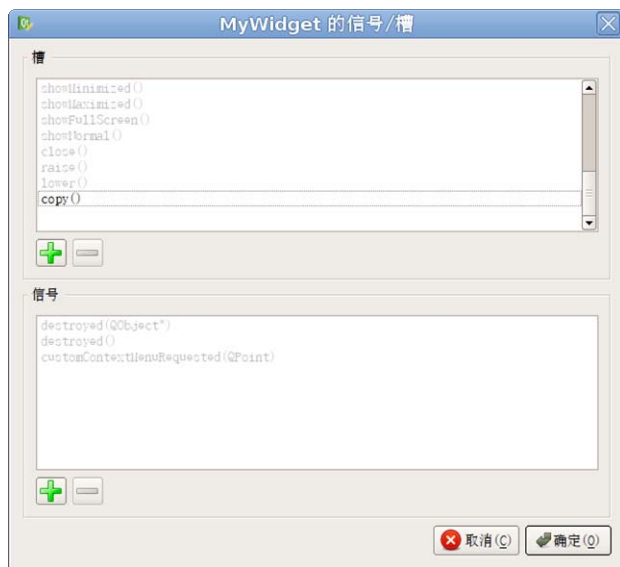


图 20.8 信号/槽编辑对话框

五、连接信号与槽

当按钮被单击时，将发射 `clicked()` 信号，所以要将 `pushButton` 部件的 `clicked()` 信号与窗体的 `copy()` 槽连接起来。要连接信号与槽，首先要将编辑模式切换到“信号/槽”模式，这可以通过“编辑”菜单中的选项来完成，也可以使用 `F4` 快捷键来完成。在“信号/槽”编辑模式下，用鼠标从信号的发送者拖动到槽的所有者，释放鼠标，即可弹出“配置连接”对话框，如图 20.9 所示。这里我们用鼠标从按钮拖动到窗体的任何位置（不能在其他部件上）以对按钮的信号和窗体的槽进行连接。

在“配置连接”对话框内，选择左边的“`clicked()`”选项和右边的“`copy()`”选项，然后单击“确定”按钮就完成了连接。连接完成之后按 `F3` 快捷键返回窗口部件编辑模式。当然也可以用信号/槽编辑器进行连接。

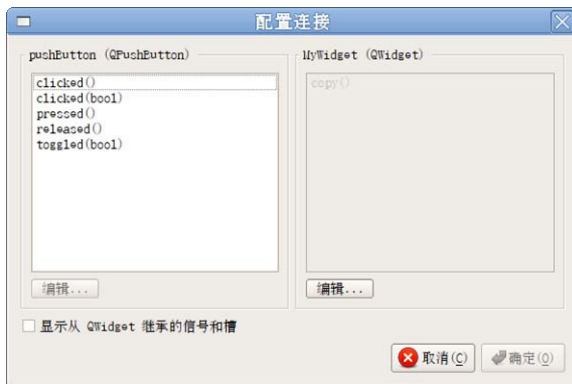


图 20.9 “配置连接”对话框

实际上, Qt 支持一种按名称连接信号与槽的方式, 它会自动将信号与名称合适的槽连接起来。这里如果我们将槽的名称命名为 `on_pushButton_clicked()`, 则 `pushButton` 的 `clicked()` 信号就会在运行时自动与这个槽连接。

至此, 整个窗体的设计就完成了, 将其保存成文件 `mywidget.ui`。按 `Ctrl+R` 组合键可以预览设计好的窗体, 如图 20.10 所示。

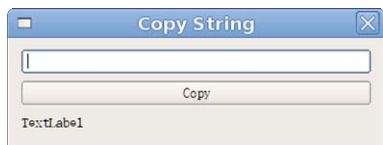


图 20.10 设计好的窗体外观

窗体设计完成之后, 对应的对象查看器和信号/槽编辑器中的内容如图 20.11 所示。

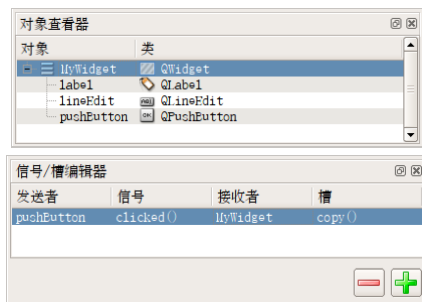


图 20.11 对象查看器和信号/槽编辑器

20.5.2 代码编写

由 `designer` 保存的文件是一个 XML 格式的文件, 它经过 `uic` 工具的编译以后会生成一个头文件, 这个头文件中定义了一个用于设置界面的类, 可统称为界面类。在这里, 由于设计界面时的窗体名称为 `MyWidget`, 所以界面类的名称为 `Ui_MyWidget`, 同时又在名字空间 `Ui` 内将其派生为另一个类 `MyWidget`, 所以也可以使用 `Ui::MyWidget`。

这个类包含了设计界面时所有部件的指针作为公共成员, 并且提供以下成员函数:

```
void setupUi(QWidget *MyWidget);
```



这个函数会为参数 `MyWidget` 指向的窗体添加所有设计时给的窗口部件及信号与槽的连接关系等。

20.5.2.1 直接使用

最简单的用法就是生成一个界面类的对象，并对需要设置界面的窗口调用其 `setupUi` 函数。为此，首先要定义一个窗口类，这个类中必须包含一个槽 `copy()`，源码如下：

```
// 文件名: mywidget.h
// 说明: 直接使用 UI 例程 MyWidget 类定义

#ifndef MYWIDGET_INCLUDED
#define MYWIDGET_INCLUDED

#include <QWidget> // 使用 QWidget 类

class MyWidget: public QWidget { // MyWidget 继承 QWidget
    Q_OBJECT
public:
    MyWidget(); // 构造函数
    virtual ~MyWidget(); // 析构函数
public slots:
    void copy(); // 声明 copy 槽
};

#endif
```

窗口类 `MyWidget` 的实现如下：

```
// 文件名: mywidget.cpp
// 说明: 直接使用 UI 例程 MyWidget 类实现

#include "mywidget.h"
#include <QLabel>
#include <QLineEdit>

MyWidget::MyWidget()
{
}

MyWidget::~~MyWidget()
{
}

void MyWidget::copy()
{
    // 查找类型为 QLabel、名字为 "label" 的子对象
    QLabel *label = findChild<QLabel *>("label");
    // 查找类型为 QLineEdit、名字为 "lineEdit" 的子对象
    QLineEdit *lineEdit = findChild<QLineEdit *>("lineEdit");
```



```

    if (label && lineEdit) {
        // 将标签的文本内容设置为编辑框的文本内容
        label->setText(lineEdit->text());
    }
}

```

代码中在 `copy()` 槽的实现里使用了 `QObject` 的模板成员函数 `findChild` 来查找需要的窗口部件。

主模块的源码如下：

```

// 文件名: main.cpp
// 说明: 直接使用 UI 例程主模块

#include <QApplication> // 使用 QApplication 类
#include "mywidget.h"    // 使用 QWidget 类
#include "ui_mywidget.h" // 使用 Ui::MyWidget 界面类

int main(int argc, char *argv[])
{
    QApplication app(argc, argv); // 构造一个 QApplication 对象
    MyWidget wdg; // 构造一个 QWidget 对象
    Ui::MyWidget ui; // 构造一个 Ui::MyWidget 对象
    ui.setupUi(&wdg); // 设置界面
    wdg.show(); // 让窗口成为显示状态
    return app.exec(); // 启动主事件循环
}

```

20.5.2.2 组合使用

直接使用界面类时，由于窗口部件的指针都定义在界面类中，在窗口类中无法访问到，因此不得不采用一些间接的方法进行访问。这样使得代码繁琐且不符合面向对象编程的一般原则。

一个可行的方法是让界面类的对象成为窗口类的成员，这样修改后的 `mywidget.h` 头文件源码如下：

```

// 文件名: mywidget.h
// 说明: 组合使用 UI 例程 MyWidget 类定义

#ifndef MYWIDGET_INCLUDED
#define MYWIDGET_INCLUDED

#include <QWidget> // 使用 QWidget 类
#include "ui_mywidget.h" // 使用 Ui::MyWidget 类

class MyWidget: public QWidget { // MyWidget 继承 QWidget
    Q_OBJECT
public:
    MyWidget(); // 构造函数
    virtual ~MyWidget(); // 析构函数
private:
    Ui::MyWidget ui; // 界面类成员
}

```



```
public slots:
    void copy(); // 声明 copy 槽
};

#endif
```

由于界面类不是 Qt 对象类，它与窗口对象之间不会有父子关系，需要自己析构，因此这里将它定义为成员变量而不是指针。注意定义为成员变量的类必须包含完整定义，而不仅仅是前向声明。

mywidget.cpp 文件的源码修改如下：

```
// 文件名: mywidget.cpp
// 说明: 组合使用 UI 例程 MyWidget 类实现

#include "mywidget.h"
#include <QLabel>
#include <QLineEdit>

MyWidget::MyWidget()
{
    ui.setupUi(this); // 设置界面
}

MyWidget::~MyWidget()
{
}

void MyWidget::copy()
{
    QLabel *label = ui.label;
    QLineEdit *lineEdit = ui.lineEdit;
    if (label && lineEdit) {
        // 将标签的文本内容设置为编辑框的文本内容
        label->setText(lineEdit->text());
    }
}
```

在这里就可以通过成员 ui 来访问各个窗口部件了。

主模块的源码如下：

```
// 文件名: main.cpp
// 说明: 组合使用 UI 例程主模块

#include <QApplication> // 使用 QApplication 类
#include "mywidget.h" // 使用 QWidget 类

int main(int argc, char *argv[])
{
    QApplication app(argc, argv); // 构造一个 QApplication 对象
    MyWidget wdg; // 构造一个 QWidget 对象
    wdg.show(); // 让窗口成为显示状态
```




```
return app.exec(); // 启动主事件循环
}
```

可以看出, 通过将界面类对象内嵌在窗口类中的方式, 窗口的实现细节被完全封装在窗口类的内部了, 窗口类的使用者无须知道窗口界面的实现方式。

使用组合方式还有一个优点就是一个窗口类可以内嵌多个不同的界面类, 从而在不同的局部或不同的时间使用不同的界面。

20.5.2.3 继承使用

通过组合方式使用界面类还有一点缺憾就是不能直接使用窗口部件的指针。可以考虑让窗口类继承界面类, 这样界面类的成员就可以在窗口类中直接使用了。当然, 窗口类还必须继承 `QWidget`, 因此是多重继承。这样修改以后, 类的定义变为:

```
class MyWidget: public QWidget, private Ui::MyWidget {
    Q_OBJECT
public:
    MyWidget(); // 构造函数
    virtual ~MyWidget(); // 析构函数
public slots:
    void copy(); // 声明 copy 槽
};
```

类的实现变为:

```
MyWidget::MyWidget()
{
    setupUi(this); // 设置界面
}

MyWidget::~MyWidget()
{
}

void MyWidget::copy()
{
    // 将标签的文本内容设置为编辑框的文本内容
    label->setText(lineEdit->text());
}
```

源码的其他部分不变。显然, 如果窗口只有一个固定的界面, 这是最简单也是最合理的使用方式。

20.5.3 运行结果

不管是哪种使用方式, 它们得到的结果都是相同的, 运行后的窗口如图 20.12 所示。



图 20.12 界面例程运行结果

20.6 Qt 常用类

使用 Qt 进行编程必须对 Qt 中常用的类有一定的了解。这一节将介绍 Qt 4.5.2 中经常用到的一些类。这些类可以分成两种：一种不是从 QObject 类派生出来的，用来表示各种基本的数据对象，如字符串、图像、字体等，这里将它们统称为基本类；另一种都是从 QWidget 类派生出来的，它们表示一个顶级窗口或者窗口部件，这里将它们统称为窗口类。

本节介绍的基本类包括 QChar, QString, QPoint, QSize, QRect, QFont, QPixmap, QIcon。

本节介绍的窗口类包括 QWidget, QDialog, QLabel, QAbstractButton, QPushButton, QCheckBox, QRadioButton, QLineEdit。

20.6.1 QChar

QChar 类是 Qt 中用于表示一个字符的类，实现在 QtCore 共享库中。QChar 类内部用 2 个字节的 Unicode 编码来表示一个字符。

20.6.1.1 构造

QChar 类提供了多个不同原型的构造函数以方便使用，如：

```
QChar(); // 构造一个空字符，即 '\0'
QChar(char ch); // 由字符型数据 ch 构造
QChar(uchar ch); // 由无符号字符型数据 ch 构造
QChar(ushort code); // 由无符号短整型数据 code 构造，code 是 Unicode 编码
QChar(short code); // 由短整型数据 code 构造，code 是 Unicode 编码
QChar(uint code); // 由无符号整型数据 code 构造，code 是 Unicode 编码
QChar(int code); // 由整型数据 code 构造，code 是 Unicode 编码
```

实际使用时很少直接构造 QChar 类的对象，而是把这些构造函数当做类型转换来用，让编译器自动构造所需的 QChar 类对象。也就是说，在所有需要 QChar 类作为参数的地方都可以安全地使用各种整数类型。

20.6.1.2 判断

QChar 类提供了很多成员函数，可以对字符的类型进行判断，如：

```
bool isDigit() const; // 判断是否是十进制数字 ('0' - '9')
bool isLetter() const; // 判断是否是字母
bool isNumber() const; // 判断是否是数字，包括正负号、小数点等
bool isLetterOrNumber() const; // 判断是否是字母或数字
bool isLower() const; // 判断是否是小写字母
```



```
bool isUpper() const; // 判断是否是大写字母
bool isNull() const; // 判断是否是空字符 '\0'
bool isPrint() const; // 判断是否是可打印字符
bool isSpace() const; // 判断是否是分隔符, 包括空格等
```

20.6.1.3 转换

QChar 类提供了一些成员函数进行数据的转换, 如:

```
char toAscii() const; // 得到字符的 ASCII 码
QChar toLower() const; // 转换成小写字母
QChar toUpper() const; // 转换成大写字母
ushort unicode() const; // 得到 Unicode 编码
```

注意这几个函数都不会改变对象自身, 转换的结果通过返回值反映出来。

20.6.1.4 比较

Qt 中还定义了一些与 QChar 类相关的比较操作符, 如:

```
bool operator!=(QChar c1, QChar c2); // 判断 c1 是否不等于 c2
bool operator<(QChar c1, QChar c2); // 判断 c1 是否小于 c2
bool operator<=(QChar c1, QChar c2); // 判断 c1 是否小于等于 c2
bool operator==(QChar c1, QChar c2); // 判断 c1 是否等于 c2
bool operator>(QChar c1, QChar c2); // 判断 c1 是否大于 c2
bool operator>=(QChar c1, QChar c2); // 判断 c1 是否大于等于 c2
```

注意这些操作符都不是 QChar 类的成员。

20.6.2 QString

QString 类是 Qt 中用于表示字符串的类, 实现在 QtCore 共享库中。QString 类在实现上有以下特征。

- ◆ 字符串采用 Unicode 内部编码, 可以表示世界上大多数语言的文字。
- ◆ 字符串的存储有引用计数, 当一个 QString 对象被复制为另一个 QString 对象时, 它们实际上指向相同的存储空间, 仅仅是增加了一个引用计数。
- ◆ 采用“按需复制”的技术, 当指向相同存储空间的多个 QString 对象中的一个要被修改时, 将真正复制一个新的字符串并进行修改。

20.6.2.1 构造

QString 类提供了很多不同原型的构造函数以方便使用, 如:

```
QString(); // 构造空字符串
QString(QChar ch); // 由 QChar 对象 ch 构造
QString(const QChar *pch, int size); // 由 QChar 数组 pch 构造, size 是数组大小
QString(const QString &obj); // 复制构造函数
QString(const char *str); // 由字符串 str 构造, str 是一个普通字符串
```

由于存在这些构造函数，凡是可以用 `QString` 类作为参数的地方，都可以安全地使用 `QChar` 对象或普通的字符串。

20.6.2.2 判断

可以用下面的成员函数判断 `QString` 对象是否为空字符串：

```
bool isEmpty() const; // 判断是否为空字符串
```

20.6.2.3 转换

`QString` 类提供了很多函数用于将字符串转换为数值，如：

```
double toDouble(bool *ok = 0) const; // 转换为高精度浮点数
float toFloat(bool *ok = 0) const; // 转换为浮点数
int toInt(bool *ok = 0, int base = 10) const; // 转换为整型数
long toLong(bool *ok = 0, int base = 10) const; // 转换为长整型数
short toShort(bool *ok = 0, int base = 10) const; // 转换为短整型数
uint toUInt(bool *ok = 0, int base = 10) const; // 转换为无符号整型数
ulong toULong(bool *ok = 0, int base = 10) const; // 转换为无符号长整型数
ushort toUShort(bool *ok = 0, int base = 10) const; // 转换为无符号短整型数
```

这些函数能够解析 `QString` 对象的内容，将其转换为相应的数值。其中 `ok` 参数指向一个 `bool` 型变量，这个参数用于输出转换是否成功的信息。`base` 参数则是转换为整数类型时的基。这些函数都不会改变 `QString` 对象自身，其用法举例如下：

```
bool ok; // 用于保存转换是否成功的信息
int hex = QString("FF").toInt(&ok, 16); // 基为 16, 结果 hex == 255, ok == true
int dec = QString("FF").toInt(&ok, 10); // 基为 10, 结果 dec == 0, ok == false
```



当字符串以 `0x` 开头时，转换的基自动为 16，当字符串以 `0` 开头时，转换的基自动为 8。

下面这些成员函数可以将一个数值转化为字符串并设为 `QString` 对象的值：

```
QString &setNum(int n, int base = 10); // 整型数
QString &setNum(uint n, int base = 10); // 无符号整型数
QString &setNum(long n, int base = 10); // 长整型数
QString &setNum(ulong n, int base = 10); // 无符号长整型数
QString &setNum(short n, int base = 10); // 短整型数
QString &setNum(ushort n, int base = 10); // 无符号短整型数
QString &setNum(double n, char format = 'g', int precision = 6); // 高精度浮点数
QString &setNum(float n, char format = 'g', int precision = 6); // 浮点数
```

将浮点数转化为字符串时，`format` 参数指定转换格式，`precision` 参数指定转换结果的精度，即有效数字的个数。注意这些函数会改变 `QString` 对象本身的值，而以下的函数则采用了不同的做法，它们返回一个新的临时对象以供使用：

```
QString number(int n, int base = 10);
```

```
QString number(uint n, int base = 10);
QString number(long n, int base = 10);
QString number(ulong n, int base = 10);
QString number(double n, char format = 'g', int precision = 6);
```

这些函数都是静态成员函数，因而与某个具体的对象无关，可以直接通过类名调用。

QString 类也提供了大小写转换的函数，如：

```
QString toLower() const; // 转换为小写
QString toUpper() const; // 转换为大写
```

这些函数都不会改变 QString 对象本身，而是将转换后的结果作为返回值。

20.6.2.4 比较

QString 类提供了一个函数用于两个 QString 对象的比较：

```
int compare(const QString &s1, const QString &s2,
            Qt::CaseSensitivity cs = Qt::CaseSensitive);
```

这是一个静态成员函数，它可以比较 s1 和 s2 的大小，参数 cs 有以下两个取值。

- ◆ Qt::CaseInsensitive：表示对大小写不敏感。
- ◆ Qt::CaseSensitive：表示对大小写敏感。

返回值的含义如下：大于 0 表示 s1 大于 s2，等于 0 表示 s1 等于 s2，小于 0 表示 s1 小于 s2。

为了方便使用，QString 类还提供了以下重载函数用于比较：

```
int compare(const QString &other,
            Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
```

这个函数用于比较 QString 对象自身和 QString 对象 other。

实际上更为直观的是使用 QString 类的比较操作符，如：

```
bool operator<(StringType other) const; // 比较是否小于 other
bool operator<=(StringType other) const; // 比较是否小于等于 other
bool operator==(StringType other) const; // 比较是否等于 other
bool operator>(StringType other) const; // 比较是否大于 other
bool operator>=(StringType other) const; // 比较是否大于等于 other
bool operator!=(StringType other) const; // 比较是否不等于 other
```

这里的 StringType 指的是 (const QString &) 或 (const char *)，也就是说，这些操作符既可以与 QString 对象比较，也可以与普通的字符串比较。它们的局限性是第一个操作数必须是 QString 对象，因此，Qt 中又定义了以下操作符：

```
bool operator<(const char *s1, const QString &s2); // 比较 s1 是否小于 s2
bool operator<=(const char *s1, const QString &s2); // 比较 s1 是否小于等于 s2
bool operator==(const char *s1, const QString &s2); // 比较 s1 是否等于 s2
bool operator>(const char *s1, const QString &s2); // 比较 s1 是否大于 s2
```



```
bool operator>=(const char *s1, const QString &s2); // 比较 s1 是否大于等于 s2
bool operator!=(const char *s1, const QString &s2); // 比较 s1 是否不等于 s2
```

这些操作符不是 `QString` 类的成员，它们的第一个参数是普通字符串。

20.6.2.5 查找

用以下的成员函数可以判断 `QString` 对象是否包含指定的字符串或字符：

```
bool contains(const QString &str,
             Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
bool contains(QChar ch,
             Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
```

用以下的成员函数可以得到 `QString` 对象包含某个特定字符串或字符的个数：

```
int count(const QString &str,
         Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
int count(QChar ch,
         Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
```

用以下的成员函数可以得到 `QString` 对象中某个特定字符串或字符出现的位置：

```
int indexOf(const QString &str, int from = 0,
           Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
int indexOf(QChar ch, int from = 0,
           Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
```

这里参数 `from` 是查找的起点，它可以为负数，`-i` 表示倒数第 `i` 个字符。查找的方向是从前往后。返回值是查找到的字符串或字符的位置，如果没有找到则返回 `-1`。

`QString` 类中还有与此功能相似的函数用于从后往前查找字符串或字符：

```
int lastIndexOf(const QString &str, int from = -1,
               Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
int lastIndexOf(QChar ch, int from = -1,
               Qt::CaseSensitivity cs = Qt::CaseSensitive) const;
```

20.6.2.6 字符串处理

`QString` 类支持用赋值操作符进行对象的复制，其赋值操作符的声明如下：

```
QString &operator=(const QString &other); // 复制另外一个 QString 对象
QString &operator=(const char *str); // 复制普通字符串
QString &operator=(char ch); // 复制字符
QString &operator=(QChar ch); // 复制 QChar 类对象
```

以下的成员函数可以将另一个字符串或字符接在 `QString` 对象后面，形成一个整体的字符串：

```
QString &append(const QString &str); // 接续 QString 对象
QString &append(const char *str); // 接续普通字符串
QString &append(QChar ch); // 接续 QChar 对象
```



它们的返回值是 `QString` 对象自己的引用,也就是说,可以用这个返回值再次调用成员函数,形成连续的字符串接续操作,如:

```
QString str;
str.append("hello").append(" ").append("QT");
```

为了让代码更直观, `QString` 类中还定义了一个操作符用于字符串的接续:

```
QString &operator+=(const QString &other); // 接续 QString 对象
QString &operator+=(const char *str); // 接续普通字符串
QString &operator+=(char ch); // 接续字符型数据
QString &operator+=(QChar ch); // 接续 QChar 对象
```

它们的功能与 `append` 相同。由于 C++ 语言允许赋值操作符和复合赋值操作符的返回值作为左值使用,因此它们的返回值也被设计为 `QString` 对象自己的引用,故也可以连续操作。但由于复合赋值操作符的结合顺序是从右往左,要想先计算左边的操作符就需要加括号,如:

```
QString str;
((str += "hello") += " ") += "QT";
```

与 `append` 函数功能类似,以下的成员函数也能够将另一个字符串或字符与 `QString` 对象连接起来,但是接在原字符串的前面:

```
QString &prepend(const QString &str); // 在前面接续 QString 对象
QString &prepend(const char *str); // 在前面接续普通字符串
QString &prepend(QChar ch); // 在前面接续 QChar 对象
```

功能更一般化的是在 `QString` 对象的任意位置插入另一个字符串或字符,如:

```
QString &insert(int position, const QString &str); // 插入字符串
QString &insert(int position, const QChar *pch, int size); // 插入 QChar 数组
QString &insert(int position, QChar ch); // 插入 QChar 对象
```

这里 `position` 参数是要插入的位置,返回值也是对 `QString` 对象自己的引用。

与插入相反的操作是移除 `QString` 对象中的一部分,如:

```
QString &remove(int position, int n);
```

这个函数可以移除 `QString` 对象中从位置 `position` 开始的 `n` 个字符,下面两个成员函数则可以从 `QString` 对象中移除指定的字符串或字符:

```
QString &remove(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive);
QString &remove(const QString &str, Qt::CaseSensitivity cs = Qt::CaseSensitive);
```

以下是 `QString` 对象的替换操作:

```
QString &replace(int position, int n, const QString &after); // QString 对象
QString &replace(int position, int n, const QChar *pch, int size); // QChar 数组
QString &replace(int position, int n, QChar after); // QChar 对象
```



这三个函数的功能是将 `QString` 对象从 `position` 开始的 `n` 个字符替换为新内容，新内容分别由 `QString` 对象、`QChar` 数组和 `QChar` 对象表示。

以下成员函数则可以搜索指定的字符串或字符并进行替换：

```
QString &replace(const QString &before, const QString &after,
               Qt::CaseSensitivity cs = Qt::CaseSensitive); // QString 替换为 QString
QString &replace(QChar ch, const QString &after,
               Qt::CaseSensitivity cs = Qt::CaseSensitive); // QChar 替换为 QString
QString &replace(QChar before, QChar after,
               Qt::CaseSensitivity cs = Qt::CaseSensitive); // QChar 替换为 QChar
```

下面这个成员函数可以清空一个 `QString` 对象的内容，使之成为空字符串：

```
void clear();
```

而下面这个成员函数可以截断 `QString` 对象，也就是去掉指定位置后的所有内容：

```
void truncate(int position); // 从位置 position 截断，位置从 0 开始编号
```

下面这个成员函数可以截掉 `QString` 对象最后的若干个字符：

```
void chop(int n); // 截掉最后的 n 个字符
```

以下几个成员函数可以得到 `QString` 对象的子字符串：

```
QString left(int n) const; // 得到左边 n 个字符构成的子字符串
QString right(int n) const; // 得到右边 n 个字符构成的子字符串
QString mid(int position, int n = -1) const; // 从中间得到子字符串
```

从中间得到子字符串时，参数 `position` 是子字符串的起始位置，参数 `n` 是字符的个数，如果 `n` 为 `-1`，则表示一直到原字符串的结尾。

注意上述三个函数并不修改 `QString` 对象自身，而是返回一个临时对象以供使用。

下面这个成员函数可以截去 `QString` 对象中头部和尾部的空白字符：

```
QString trimmed() const;
```

空白字符包括空格、回车、换行、制表符等字符。下面这个成员函数不仅能去掉 `QString` 对象头尾的空白字符，还能将中间的连续多个空白字符全部替换成一个空格：

```
QString simplified() const;
```

加法操作符可以将两个字符串或字符连接起来并以 `QString` 临时对象的方式返回：

```
const QString operator+(const QString &s1, const QString &s2);
const QString operator+(const QString &s1, const char *s2);
const QString operator+(const char *s1, const QString &s2);
const QString operator+(char ch, const QString &s);
const QString operator+(const QString &s, char ch);
```

注意加法操作符的两个操作数中必须至少有一个是 `QString` 对象，否则无法重载操作符。显然，加法操作符都不是 `QString` 类的成员。



20.6.2.7 索引

`QString` 类也像普通的字符串一样可以根据下标得到某个位置上的字符：

```
const QChar at(int position) const;
```

这是一个成员函数，更直观的方法是用以下的操作符：

```
const QChar operator[](int position) const;
const QChar operator[](uint position) const;
```

这样对 `QString` 对象的取字符操作就类似于对一个字符数组的操作。事实上，通过 `[]` 操作符得到的字符还可以被修改，这要用到另外两个重载的 `[]` 操作符：

```
QCharRef operator[](int position);
QCharRef operator[](uint position);
```

返回的 `QCharRef` 类是一个辅助类，对它的修改将反映到原字符串中去。

20.6.2.8 统计

以下两个成员函数都可以得到 `QString` 对象中字符的个数：

```
int size() const;
int length() const;
```

注意字符的个数并不一定等于字节数。

20.6.3 QPoint

`QPoint` 类代表一个坐标点，实现在 `QtCore` 共享库中。它可以认为是一个整型的横坐标和一个整型的纵坐标的组合。

20.6.3.1 构造

`QPoint` 类支持以下两种构造方式：

```
QPoint(); // 构造横纵坐标均为 0 的 QPoint 对象
QPoint(int x, int y); // 构造横纵坐标分别为 x 和 y 的 QPoint 对象
```

20.6.3.2 属性

通过以下成员函数可得到 `QPoint` 对象中的横纵坐标的引用：

```
int &rx(); // 得到横坐标的引用
int &ry(); // 得到纵坐标的引用
```

注意这些引用都不是只读的，也就是说可以通过它们直接修改 `QPoint` 对象的横纵坐标。通过以下的成员函数可以设置 `QPoint` 对象中的横纵坐标：

```
void setX(int x); // 设置横坐标为 x
void setY(int y); // 设置纵坐标为 y
```

下面两个成员函数则是只读的，可以获得 `QPoint` 对象中的横纵坐标：

```
int x() const; // 获得横坐标
int y() const; // 获得纵坐标
```

20.6.3.3 操作符

`QPoint` 类支持加法和减法的复合赋值操作：

```
QPoint &operator+=(const QPoint &point); // 加赋值
QPoint &operator-=(const QPoint &point); // 减赋值
```

这两个操作符是它的成员。而以下的操作符则不是它的成员：

```
const QPoint operator+(const QPoint &p1, const QPoint &p2); // 加法
const QPoint operator-(const QPoint &p1, const QPoint &p2); // 减法
const QPoint operator-(const QPoint &point); // 取负数
bool operator==(const QPoint &p1, const QPoint &p2); // 判断是否相等
bool operator!=(const QPoint &p1, const QPoint &p2); // 判断是否不等
```

加法、减法、取负数操作都相当于对横纵坐标分别进行操作。

20.6.4 QSize

`QSize` 类代表一个矩形区域的大小，实现在 `QtCore` 共享库中。它可以认为是由一个整型的宽度和整型的高度组合而成的。

20.6.4.1 构造

```
QSize(); // 构造一个非法的 QSize 对象
QSize(int width, int height); // 构造宽度为 width、高度为 height 的 QSize 对象
```

20.6.4.2 属性

以下成员函数可以得到 `QSize` 对象中宽度和高度的引用：

```
int &rwidth(); // 得到宽度的引用
int &rheight(); // 得到高度的引用
```

通过返回的引用可以直接修改 `QSize` 对象的宽度和高度。

通过以下成员函数可以设置 `QSize` 对象的宽度和高度：

```
void setWidth(int width); // 设置宽度
void setHeight(int height); // 设置高度
```

通过以下成员函数可以获得 `QSize` 对象的宽度和高度：

```
int width() const; // 获得宽度
int height() const; // 获得高度
```

20.6.4.3 操作符

QSize 类支持和 QPoint 类相似的操作符，不再赘述。

20.6.5 QRect

QRect 类代表一个矩形区域，实现在 QtCore 共享库中。它可以认为是一个 QPoint 对象和一个 QSize 对象的组合，QPoint 对象是它的左上角的坐标，QSize 对象则是它的大小。

20.6.5.1 构造

无参数的构造函数 QRect() 将构造一个高度和宽度都为 0 的矩形区域。而以下的构造函数则可以指定坐标和大小：

```
QRect(const QPoint &topLeft, const QSize &size);
```

这个函数将构造一个左上角坐标为 topLeft、大小为 size 的矩形区域。另外一个可选的构造函数如下：

```
QRect(int x, int y, int width, int height);
```

这个函数将构造一个左上角横纵坐标分别为 x 和 y、宽度为 width、高度为 height 的矩形区域，等价于：

```
QRect(QPoint(x, y), QSize(width, height));
```

20.6.5.2 属性

QRect 类的常用属性如表 20.3 所示。

表 20.3 QRect 类常用属性

属性含义	获取属性的成员函数	设置属性的成员函数
左上角横坐标	int left() const;	void setLeft(int x);
左上角纵坐标	int top() const;	void setTop(int y);
左上角横坐标	int x() const;	void setX(int x);
左上角纵坐标	int y() const;	void setY(int y);
宽度	int width() const;	void setWidth(int width);
高度	int height() const;	void setHeight(int height);
左上角坐标	QPoint topLeft() const;	void setTopLeft(const QPoint &position);
大小	QSize size() const;	void setSize(const QSize &size);

这里需要注意一点，当重新设置左上角坐标时，矩形区域的右下角坐标保持不变，也就是说宽度和高度有可能随之而变。如果要同时改变矩形区域的所有属性，可以用下面这个成员函数：

```
void setRect(int x, int y, int width, int height); // 同时设置矩形区域的 4 个属性
```

如果要移动矩形区域的位置，也就是说重设左上角坐标，但不改变宽度和高度，则应该用以下

系列函数：

```
void moveLeft(int x); // 左上角横坐标移动到 x
void moveTop(int y); // 左上角纵坐标移动到 y
void moveTopLeft(const QPoint &position); // 左上角坐标移动到 position
void moveTo(int x, int y); // 左上角坐标移动到 (x, y)
void moveTo(const QPoint &position); // 左上角坐标移动到 position
```

20.6.5.3 操作符

按位与操作符可以求两个 `QRect` 对象的相交区域，定义如下：

```
QRect operator&(const QRect &rectangle) const; // 求相交区域
QRect &operator&=(const QRect &rectangle); // 求相交区域后赋值
```

按位或操作符可以得到能够同时包含两个 `QRect` 对象的最小矩形区域，定义如下：

```
QRect operator|(const QRect &rectangle) const; // 求合并区域
QRect &operator|=(const QRect &rectangle); // 求合并区域后赋值
```

当然，`QRect` 类也支持进行相等与否判断的比较操作符：

```
bool operator!=(const QRect &r1, const QRect &r2); // 判断是否不等
bool operator==(const QRect &r1, const QRect &r2); // 判断是否相等
```

20.6.6 QFont

`QFont` 类代表字体，实现在 `QtGui` 共享库中。

20.6.6.1 构造

`QFont` 类有以下几个常用的构造函数：

```
QFont(); // 由应用程序的默认字体构造新字体对象
QFont(const QString &family, int pointSize = -1,
       int weight = -1, bool italic = false);
```

其中第二个构造函数的各个参数的含义解释如下。

- ◆ `family`：字体的名称。
- ◆ `pointSize`：字体的点大小，如果这个参数小于等于 0，则自动设为 12。
- ◆ `weight`：字体的粗细。
- ◆ `italic`：字体是否为斜体。

这些参数也可以在字体对象构造以后通过属性来修改。

20.6.6.2 属性

`QFont` 类的常用属性如表 20.4 所示。

表 20.4 QFont 类的常用属性

字体的属性	获取所用成员函数	设置所用成员函数
名称	<code>QString family() const;</code>	<code>void setFamily(const QString &family);</code>
点大小	<code>int pointSize() const;</code>	<code>void setPointSize(int pointSize);</code>
像素大小	<code>int pixelSize() const;</code>	<code>void setPixelSize(int pixelSize);</code>
粗细	<code>int weight() const;</code>	<code>void setWeight(int weight);</code>
粗体	<code>bool bold() const;</code>	<code>void setBold(bool enable);</code>
斜体	<code>bool italic() const;</code>	<code>void setItalic(bool enable);</code>
下画线	<code>bool underline() const;</code>	<code>void setUnderline(bool enable);</code>

其中设置粗体属性实际上就是将字体的粗细设为一个特定的值。点大小与像素大小是指定字体大小的两种方式。如果指定了点大小，则像素大小属性的值就是 -1；反之如果指定了像素大小，则点大小属性的值就是 -1。

如果指定的字体在使用时没有对应的字体文件，Qt 将自动选择最接近的字体。如果要显示的字符在字体中不存在，则字符会被显示为一个空心方框。

20.6.7 QPixmap

QPixmap 类代表图像，实现在 QtGui 共享库中。

20.6.7.1 构造

以下构造函数生成的 QPixmap 对象为空图像：

```
QPixmap(); // 构造一个大小为 0 的空图像
```

以下构造函数生成指定大小的 QPixmap 对象，但图像数据未初始化：

```
QPixmap(const QSize &size); // 构造大小为 size 的图像，图像数据未初始化
QPixmap(int width, int height); // 等价于 QPixmap(QSize(width, height));
```

以下构造函数能够从指定的文件中加载图像并生成 QPixmap 对象：

```
QPixmap(const QString &fileName, const char *format = 0,
        Qt::ImageConversionFlags flags = Qt::AutoColor);
```

其各个参数的含义解释如下。

- ◆ fileName：文件名。
- ◆ format：字符串，表示图像文件的格式，如果为 0，将进行自动识别。
- ◆ flags：表示颜色的转换模式。

如果图像文件加载失败则产生空图像。这里 flags 参数有以下取值。

- ◆ Qt::AutoColor：由系统自动决定。
- ◆ Qt::ColorOnly：彩色模式。

◆ Qt::MonoOnly: 单色模式。

20.6.7.2 图像参数

以下成员函数可以获得 QPixmap 对象所表示的图像的相关信息:

```
int depth() const; // 颜色深度, 即每像素所占的比特数
int width() const; // 图像宽度, 单位是像素
int height() const; // 图像高度, 单位是像素
QSize size() const; // 图像的大小, 即 QSize(width(), height());
QRect rect() const; // 图像的矩形区域, 即 QRect(QPoint(0, 0), size());
```

20.6.7.3 加载和保存图像

用下面的成员函数可以从文件加载图像:

```
bool load(const QString &fileName, const char *format = 0,
          Qt::ImageConversionFlags flags = Qt::AutoColor);
```

这里各个参数的含义与构造函数中一样, 返回值为 true 表示加载成功, false 表示加载失败。相反的操作是将 QPixmap 代表的图像保存到文件, 可用以下成员函数:

```
bool save(const QString &fileName, const char *format = 0,
          int quality = -1) const;
```

其各个参数及返回值的含义解释如下。

- ◆ fileName: 文件名。
- ◆ format: 字符串, 表示图像文件的格式, 如果为 0, 将根据文件名的后缀自动确定文件格式。
- ◆ quality: 对于有损压缩的文件格式来说, 它表示图像保存的质量, 质量越低压缩率越大。取值范围为 0~100, -1 表示采用默认值。
- ◆ 返回值: true 表示保存成功, false 表示保存失败。

20.6.7.4 判断

以下成员函数可以判断 QPixmap 对象是否为空图像:

```
bool isNull() const; // 判断是否为空图像
```

20.6.8 QIcon

QIcon 类代表图标, 实现在 QtGui 共享库中。QIcon 对象可以认为是一系列图像的组合, 每个图像代表窗口在某种状态下应该显示的图标。

20.6.8.1 构造

QIcon 类支持以下构造函数:

```
QIcon(); // 构造一个空图像构成的图标
QIcon(const QPixmap &pixmap); // 从 QPixmap 对象构造图标
```

```
QIcon(const QString &fileName); // 从图像文件构造图标
```

当从 `QPixmap` 对象构造图标时，系统会自动产生窗口不同状态下对应的图像，比如窗口在禁用状态下其图标为灰色。从文件构造图标时，文件并不是立刻加载，而是当图标要显示时才加载。

20.6.8.2 添加图像

下面的成员函数可以从图像文件中向 `QIcon` 对象添加图像：

```
void addFile(const QString &fileName, const QSize &size = QSize(),
            Mode mode = Normal, State state = Off);
```

其中各个参数的含义解释如下。

- ◆ `fileName`：文件名。
- ◆ `size`：指定大小。
- ◆ `mode`：指定使用模式，即窗口在何种状态下使用这个图像。
- ◆ `state`：指定使用状态。

`Mode` 为 `QIcon` 类的成员类型，有以下取值。

- ◆ `QIcon::Normal`：窗口为使能状态，但未激活。
- ◆ `QIcon::Disabled`：窗口为禁用状态。
- ◆ `QIcon::Active`：窗口为激活状态。
- ◆ `QIcon::Selected`：窗口被选中。

当窗口的状态切换时，默认的图标绘制函数会自动根据窗口的状态重绘图标。如果窗口还有所谓的开关状态（比如一个按钮可以有按下和弹起两个状态），则还可以根据 `state` 参数来选择不同的图像。`state` 参数为 `State` 类型，这也是一个 `QIcon` 类的成员类型，它有以下取值。

- ◆ `QIcon::Off`：窗口在关状态。
- ◆ `QIcon::On`：窗口在开状态。

另外一个成员函数可以直接将 `QPixmap` 对象添加到 `QIcon` 对象中：

```
void addPixmap(const QPixmap &pixmap,
              Mode mode = Normal, State state = Off);
```

这里的 `pixmap` 参数是要添加的 `QPixmap` 对象，`mode` 参数和 `state` 参数的含义与 `addFile` 函数中相同。

20.6.8.3 获取图像

下面的成员函数可以获取 `QIcon` 对象中的图像：

```
QPixmap pixmap(const QSize &size, Mode mode = Normal, State state = Off) const;
```

其中参数 `size` 是指定的大小，参数 `mode` 和 `state` 则是使用模式和状态。这个函数还有以下的重载版本：

```
QPixmap pixmap(int w, int h, Mode mode = Normal, State state = Off) const;
QPixmap pixmap(int extent, Mode mode = Normal, State state = Off) const;
```

它们分别等价于以下的函数调用：

```
pixmap(QSize(w, h), mode, state);
pixmap(QSize(extent, extent), mode, state);
```

注意返回的图像大小可能比指定的小，但不会比指定的大。

20.6.8.4 判断

以下成员函数可以判断 `QIcon` 对象是否为空图像构成的图标：

```
bool isNull() const; // 判断是否为空图像构成的图标
```

20.6.9 QWidget

`QWidget` 类代表一般的窗口，其他窗口类都是从 `QWidget` 类继承出来的。而 `QWidget` 类则同时继承了 `QObject` 类和 `QPaintDevice` 类，也就是说，窗口类都是 Qt 对象类。这里的 `QPaintDevice` 类则是所有可绘制的对象的基类。

如图 20.13 所示为常用窗口类的继承关系。

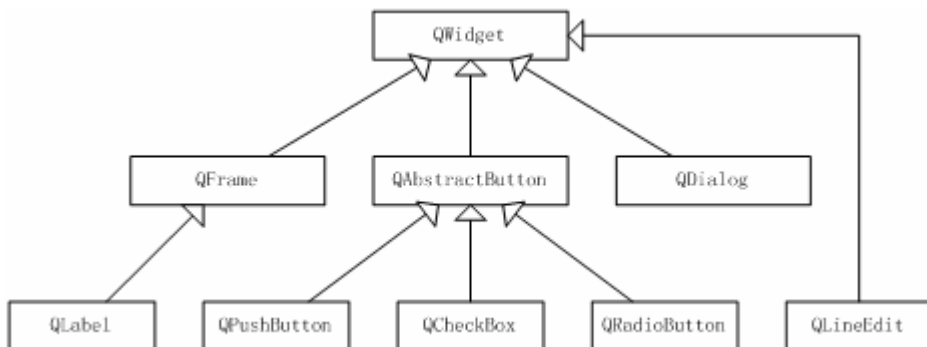


图 20.13 常用窗口类的继承关系

所有的窗口类都与 GUI 有关，因此都实现在 `QtGui` 共享库中。

20.6.9.1 构造

`QWidget` 类的构造函数如下：

```
QWidget(QWidget *parent = 0, Qt::WindowFlags f = 0);
```

其中参数 `parent` 指向父窗口，如果这个参数为 `0`，则窗口就成为一个顶级窗口。

参数 `f` 是构造窗口的标志，主要用于控制窗口的类型和外观等，有以下常用值。

- ◆ `Qt::FramelessWindowHint`：没有边框的窗口。
- ◆ `Qt::WindowStaysOnTopHint`：总在最上面的窗口。
- ◆ `Qt::CustomizeWindowHint`：自定义窗口标题栏，以下标志必须与这个标志一起使用才有效，

否则窗口将有默认的标题栏。

- ◆ Qt::WindowTitleHint: 显示窗口标题栏。
- ◆ Qt::WindowSystemMenuHint: 显示系统菜单。
- ◆ Qt::WindowMinimizeButtonHint: 显示最小化按钮。
- ◆ Qt::WindowMaximizeButtonHint: 显示最大化按钮。
- ◆ Qt::WindowMinMaxButtonsHint: 显示最小化按钮和最大化按钮。
- ◆ Qt::WindowCloseButtonHint: 显示关闭按钮。

20.6.9.2 独立窗口

窗口构造的时候如果有 Qt::Window 标志, 那么它就是一个独立窗口, 否则就是一个依附于其他独立窗口的窗口部件。顶级窗口一定是独立窗口, 但独立窗口不一定是顶级的, 它可以有父窗口, 当父窗口被析构时它也会随之被析构。独立窗口一般有自己的外边框和标题栏, 可以有移动、改变大小等操作。

一个窗口是否为独立窗口可用下面的成员函数来判断:

```
bool isWindow() const; // 判断是否为独立窗口
```

下面这个函数可以得到窗口部件所在的独立窗口:

```
QWidget *window() const; // 得到所在的独立窗口
```

当然, 如果窗口本身就是独立窗口, 那么得到的就是自己。

而下面这个函数可以得到窗口的父窗口:

```
QWidget *parentWidget() const; // 得到父窗口
```

20.6.9.3 窗口标题

windowTitle 属性表示窗口的标题, 与之相关的成员函数如下:

```
QString windowTitle() const; // 获得窗口标题
void setWindowTitle(const QString &text); // 设置窗口标题为 text
```

20.6.9.4 几何参数

这里的几何参数指的是窗口的大小和位置。一个窗口有两套几何参数, 一套是窗口外边框所占的矩形区域, 另一套是窗口客户区所占的矩形区域。所谓窗口客户区就是窗口中除去边框和标题栏用来显示内容的区域。两套几何参数的示意如图 20.14 所示。

这两套几何参数分别由两个 QRect 型的属性代表, 相关的成员函数如下:

```
const QRect &geometry() const; // 获取客户区几何参数
void setGeometry(int x, int y, int w, int h); // 设置客户区几何参数
void setGeometry(const QRect &rect); // 设置客户区几何参数
QRect frameGeometry() const; // 获取外边框几何参数
```

这里虽然没有直接设置外边框几何参数的函数, 但客户区几何参数变化之后, 外边框的几何参数也会随之变化。设置几何参数可能会使窗口的位置及大小发生变化, 这时会发送窗口移动事件

QMoveEvent，如果大小有变化，还会发送窗口改变大小事件 **QResizeEvent**，事件的处理函数分别是 **moveEvent** 和 **resizeEvent**。注意这里的坐标都是相对于父窗口的，因此移动一个窗口并不导致它的所有部件都收到移动事件。

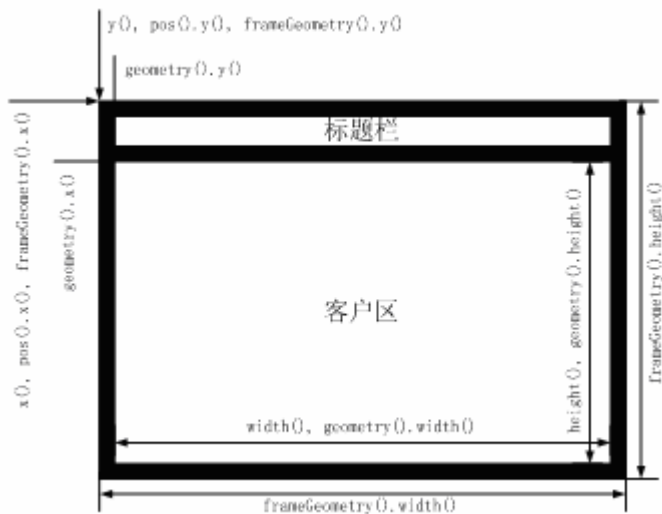


图 20.14 窗口的几何参数



不要在 **moveEvent** 或 **resizeEvent** 两个事件处理函数中设置几何参数，否则将导致无限循环。

窗口的几何参数也可以由用户的操作改变，这时也会发送相应的事件。

为了方便使用，与几何参数相关的成员函数还有以下这些：

```
QPoint pos() const; // 获得窗口左上角的坐标（外边框几何参数）
QSize size() const; // 窗口大小（客户区几何参数）
int x() const; // 窗口左上角横坐标（外边框几何参数）
int y() const; // 窗口左上角纵坐标（外边框几何参数）
int height() const; // 窗口高度（客户区几何参数）
int width() const; // 窗口宽度（客户区几何参数）
```

可以看出，坐标全部是外边框几何参数，而大小全部是客户区几何参数。要获得外边框的大小需要用下面这个成员函数：

```
QSize frameSize() const; // 窗口大小（外边框几何参数）
```

改变这些属性可以用下面这些成员函数：

```
void move(int x, int y); // 将窗口左上角移动到坐标 (x, y) 处
void move(const QPoint &pos); // 将窗口左上角移动到 pos 处
void resize(int w, int h); // 将窗口的宽度改为 w，高度改为 h
void resize(const QSize &size); // 将窗口的大小改为 size
```

同样，这里 **move** 函数用的是外边框几何参数，而 **resize** 函数用的是客户区几何参数。

还有一个属性比较特殊, 相关的成员函数如下:

```
QRect rect() const; // 获取窗口客户区域
```

它获得的坐标都是相对于窗口自己的客户区的, 也就是说横纵坐标永远是 0。



对于一个窗口部件来说, 它的两套几何参数是一致的。

20.6.9.5 可见性与隐藏

可见性指的是窗口是否显示在屏幕上的属性。被其他窗口暂时遮挡的窗口也属于可见的。可见性由窗口的 **visible** 属性表示, 与之相关的成员函数如下:

```
bool isVisible() const; // 判断窗口是否可见
bool isHidden() const; // 判断窗口是否隐藏
virtual void setVisible(bool visible); // 设置窗口是否隐藏
void setHidden(bool hidden); // 等价于 setVisible(!hidden)
```

visible 属性为 **true** 时表示窗口可见, 为 **false** 时表示窗口不可见。这里要注意的是, **setVisible** 函数实际上设置的是窗口是否隐藏, 而不是可见性。可见性与隐藏有如下关系。

- ◆ 隐藏的窗口一定是不可见的。
- ◆ 非隐藏的窗口在它的父窗口可见的情况下也是可见的。
- ◆ 非隐藏的顶级窗口是可见的。

setVisible 和 **setHidden** 同时也是槽, 它们一般并不直接使用, 而是使用以下几个槽:

```
void show(); // 显示窗口, 等价于 setVisible(true);
void hide(); // 隐藏窗口, 等价于 setHidden(true);
```

当窗口显示时, 将发送 **QShowEvent** 事件; 当窗口隐藏时, 将发送 **QHideEvent** 事件。事件的处理函数分别是 **showEvent** 和 **hideEvent**。

20.6.9.6 窗口状态

独立窗口有正常、全屏、最大化、最小化几种状态, 与之相关的成员函数如下:

```
bool isMinimized() const; // 判断窗口是否为最小化
bool isMaximized() const; // 判断窗口是否为最大化
bool isFullScreen() const; // 判断窗口是否为全屏
void showMinimized(); // 以最小化方式显示窗口, 这是一个槽
void showMaximized(); // 以最大化方式显示窗口, 这是一个槽
void showFullScreen(); // 以全屏方式显示窗口, 这是一个槽
void showNormal(); // 以正常方式显示窗口, 这是一个槽
```

注意后 4 个函数同时也是槽。全屏方式与最大化方式的区别在于: 全屏方式下窗口的边框和标题栏消失, 客户区占据整个屏幕。窗口的各种状态仅对独立窗口有效, 对窗口部件来说没有意义。

另外还有一个 **windowState** 属性与窗口状态有关, 相关的成员函数如下:

```
Qt::WindowStates windowState() const; // 获取窗口状态
void setWindowState(Qt::WindowStates windowState); // 设置窗口状态
```

这里的 `Qt::WindowStates` 类型有以下几个取值。

- ◆ `Qt::WindowNoState`: 无标志, 正常状态。
- ◆ `Qt::WindowMinimized`: 最小化状态。
- ◆ `Qt::WindowMaximized`: 最大化状态。
- ◆ `Qt::WindowFullScreen`: 全屏状态。
- ◆ `Qt::WindowActive`: 激活状态。

这些取值可以用“按位或”的方法组合起来使用。

需要注意的是, 调用 `setWindowState` 函数将使窗口变为隐藏状态。

20.6.9.7 使能

处于使能状态的窗口才能处理键盘和鼠标等输入事件, 反之, 处于禁用状态的窗口不能处理这些事件。窗口是否处于使能状态由属性 `enabled` 表示, 相关成员函数如下:

```
bool isEnabled() const; // 获得窗口的使能状态
void setEnabled(bool enable); // 设置窗口的使能状态, 这是一个槽
void setDisabled(bool disable); // 等价于 setEnabled(!disable), 这是一个槽
```

其中两个设置属性的函数同时也是槽。窗口的使能状态也可能影响外观, 比如处于禁用状态的按钮文本显示为灰色。

使能状态与窗口的可见性有相似的逻辑: 禁用一个窗口同时会使它的所有子窗口成为禁用状态。

20.6.9.8 激活状态

当有多个独立窗口同时存在时, 只有一个窗口能够处于激活状态。系统产生的键盘、鼠标等输入事件将被发送给处于激活状态的窗口。一般来说, 这样的窗口会被提升到堆叠层次的最上面, 除非其他窗口有总在最上面的属性。与激活状态相关的成员函数如下:

```
bool isActiveWindow() const; // 判断窗口所在的独立窗口是否激活
void activateWindow(); // 设置窗口所在的独立窗口为激活状态
```

注意这里操作的其实不是窗口本身, 而是窗口所在的独立窗口, 因为窗口部件是没有激活状态的概念的。

20.6.9.9 焦点

焦点用来控制同一个独立窗口内哪一个部件可以接受键盘事件, 同一时刻只能有一个部件获得焦点。与焦点有关的成员函数如下:

```
bool hasFocus() const; // 判断窗口是否获得焦点
void setFocus(); // 使窗口获得焦点, 这是一个槽
```

```
void clearFocus(); // 使窗口失去焦点
QWidget *focusWidget() const; // 得到窗口内获得焦点的子窗口
```

setFocus 函数同时又是一个槽。窗口部件得到焦点以后，别忘了还需要它所在的独立窗口处于激活状态才能得到键盘事件。

一个窗口获得焦点，同时意味着另一个窗口失去焦点。当窗口获得或失去焦点时，将发送 **QFocusEvent** 事件，它有两个处理函数：**focusInEvent** 和 **focusOutEvent**，分别对应获得焦点和失去焦点。

值得一提的是 **editFocus** 属性，这是一个专门用于嵌入式系统的属性。因为嵌入式系统通常键盘较小，没有专门用于切换焦点的 **Tab** 键，所以上下方向键被用来切换焦点。如果一个窗口部件设置 **editFocus** 属性为 **true**，则上下方向键就不再用来切换焦点，而是发送给这个窗口。与这个属性相关的成员函数如下：

```
bool hasEditFocus() const; // 判断窗口是否有 editFocus 属性
void QWidget::setEditFocus(bool enable); // 设置窗口的 editFocus 属性
```

20.6.9.10 捕获键盘和鼠标事件

窗口部件即使获得焦点，也不一定能获得键盘事件，因为其他窗口可能会捕获键盘事件。捕获了键盘事件的窗口将得到所有键盘事件，而其他窗口将完全得不到键盘事件，直到捕获了键盘事件的窗口释放键盘事件。与键盘事件捕获相关的成员函数如下：

```
void grabKeyboard(); // 捕获键盘事件
void releaseKeyboard(); // 释放键盘事件
```

类似的还有鼠标事件的捕获和释放，其成员函数如下：

```
void grabMouse(); // 捕获鼠标事件
void releaseMouse(); // 释放鼠标事件
```

对键盘事件和鼠标事件的捕获是相互独立的。这里要注意两点：一是如果有另外一个窗口进行了捕获操作，则当前处于捕获状态的窗口将失去对事件的捕获；二是只有可见的窗口才能进行输入事件捕获。

以下的成员函数能够得到应用程序中正在捕获键盘或鼠标事件的窗口：

```
QWidget *keyboardGrabber(); // 得到正在捕获键盘事件的窗口
QWidget *mouseGrabber(); // 得到正在捕获鼠标事件的窗口
```

这两个函数是静态函数。

20.6.9.11 布局

属性 **layout** 代表窗口的顶级布局，相关的成员函数如下：

```
QLayout *layout() const; // 获得顶级布局
void setLayout(QLayout *layout); // 设置顶级布局
```

20.6.9.12 字体

font 属性表示所用的字体，相关的成员函数如下：

```
const QFont &font() const; // 获得字体
void setFont(const QFont &); // 设置字体
```

如果没有为窗口设置字体，则窗口自动使用父窗口的字体，顶级窗口则使用应用程序的默认字体。

20.6.9.13 信号

当窗口要被析构时会发射以下信号：

```
void destroyed(QObject *obj = 0);
```

这是一个从 **QObject** 类继承过来的信号。**QObject** 对象析构时，先发射这个信号，然后才析构它的所有子对象。

20.6.9.14 槽

在前面的介绍中已经提及了 **QWidget** 类的许多槽，这里将介绍其他常用的槽。

下面的槽可以关闭窗口：

```
bool close();
```

当这个槽被调用时，首先向窗口发送一个关闭事件，如果事件被接受，则窗口隐藏，如果被拒绝，则什么也不做。如果窗口设置了 **Qt::WA_QuitOnClose** 属性，则窗口对象会被析构，大多数类型的窗口都默认设置了这个属性。

这个槽的返回值表示关闭事件是否被接受，也就是窗口是否真的被关闭了。

下面的槽可以提升或降低窗口所在的堆叠层次：

```
void lower(); // 降低窗口到最下面
void raise(); // 提升窗口到最上面
```

这里窗口的堆叠层次仅局限在同一个父窗口内，也就是说，只有各个兄弟窗口之间才有互相遮挡的情况发生。

20.6.9.15 事件

QWidget 类能够处理类型丰富的事件，这里将介绍一些常用的事件处理函数。

窗口事件：

```
virtual void closeEvent(QCloseEvent *event); // 关闭
virtual void showEvent(QShowEvent *event); // 显示
virtual void hideEvent(QHideEvent *event); // 隐藏
virtual void moveEvent(QMoveEvent *event); // 移动
virtual void resizeEvent(QResizeEvent *event); // 改变大小
```

这里通过 `QMoveEvent` 类的以下成员函数可以获得窗口的旧坐标和新坐标：

```
const QPoint &oldPos() const; // 旧坐标
const QPoint &pos() const; // 新坐标
```

通过 `QResizeEvent` 类的以下成员函数可以获得窗口的旧大小和新大小：

```
const QSize &oldSize() const; // 旧大小
const QSize &size() const; // 新大小
```

键盘事件：

```
virtual void keyPressEvent(QKeyEvent *event); // 键按下
virtual void keyReleaseEvent(QKeyEvent *event); // 键松开
```

这里通过 `QKeyEvent` 类的成员函数可以获得关于按键的一些信息，如：

```
int key() const; // 得到键值
```

鼠标事件：

```
virtual void mousePressEvent(QMouseEvent *event); // 鼠标键按下
virtual void mouseReleaseEvent(QMouseEvent *event); // 鼠标键松开
virtual void mouseDoubleClickEvent(QMouseEvent *event); // 鼠标键双击
virtual void mouseMoveEvent(QMouseEvent *event); // 鼠标移动
virtual void enterEvent(QEvent *event); // 鼠标进入窗口
virtual void leaveEvent(QEvent *event); // 鼠标离开窗口
virtual void wheelEvent(QWheelEvent *event); // 鼠标滚轮移动
```

这里通过 `QMouseEvent` 事件的成员函数可获得关于鼠标的信息，如：

```
const QPoint &pos() const; // 得到鼠标坐标（相对于接收事件的窗口）
int x() const; // 得到鼠标横坐标（相对于接收事件的窗口）
int y() const; // 得到鼠标纵坐标（相对于接收事件的窗口）
const QPoint &globalPos() const; // 得到鼠标坐标（全局坐标）
int globalX() const; // 得到鼠标横坐标（全局坐标）
int globalY() const; // 得到鼠标纵坐标（全局坐标）
Qt::MouseButton button() const; // 得到引起事件的鼠标键
Qt::MouseButtons buttons() const; // 得到事件发生时的鼠标键状态
```

其中 `Qt::MouseButton` 是一个枚举类型，有以下常用取值。

- ◆ `Qt::NoButton`：无键。
- ◆ `Qt::LeftButton`：左键。
- ◆ `Qt::RightButton`：右键。
- ◆ `Qt::MidButton`：中键。

注意，对于鼠标移动事件 `QMouseEvent` 的 `button` 函数总是返回 `Qt::NoButton`，而 `buttons` 函数的返回值则是 `Qt::MouseButton` 类型的“按位或”组合，它能反映事件发生时鼠标键的按下状态。

`QWheelEvent` 类则代表滚轮事件，它有一套与 `QMouseEvent` 类几乎相同的成员函数，但少一个 `button` 函数，多以下两个函数：

```
int delta () const; // 获得滚轮转动的角度
Qt::Orientation orientation () const; // 获得滚轮转动的方向
```

其中 `Qt::Orientation` 是一个枚举类型，它有以下取值。

- ◆ `Qt::Horizontal`：横向。
- ◆ `Qt::Vertical`：纵向。

焦点事件：

```
virtual void focusInEvent(QFocusEvent *event); // 获得焦点
virtual void focusOutEvent(QFocusEvent *event); // 失去焦点
```

这些事件处理函数都没有返回值，因此如果要接受或拒绝一个事件就要调用 `QEvent` 类的成员函数，如：

```
event->accept(); // 接受事件
event->ignore(); // 拒绝事件
```

事件被拒绝后的结果视具体情况而定，比如关闭事件被拒绝后，窗口将不会被关闭，而键盘、鼠标等输入事件被拒绝后会向上传播到父窗口。

20.6.10 QDialog

`QDialog` 类代表对话框。对话框一般用来实现那些只是暂时存在的用户界面。对话框是独立的窗口，但通常它也有父窗口，当对话框显示时，默认的位置在父窗口的中央。从外观上来看，对话框一般没有最大化、最小化按钮。

对话框有模态和非模态两种形式。非模态对话框的行为和使用方法都类似于普通的窗口。模态对话框则有所不同，当模态对话框显示时，其他窗口将全部进入非激活状态，不能接受键盘和鼠标事件。模态的方式又可以分为两种，一种是对整个应用程序模态，这时它的出现将导致程序中的所有窗口失去响应；另一种是对窗口模态，这时仅仅会导致它所在的整个窗口树失去响应。

20.6.10.1 构造

`QDialog` 类的构造函数与 `QWidget` 类形式相同：

```
QDialog(QWidget *parent = 0, Qt::WindowFlags f = 0);
```

20.6.10.2 模态性

用下面的成员函数可将对话框设为模态：

```
void setModal(bool modal); // 设置对话框的模态性
```

当参数 `modal` 为 `true` 时，对话框设为模态，否则设为非模态。如果设为模态，则默认是对

整个应用程序模态的。要设为对窗口模态，则需要用以下函数：

```
void setWindowModality(Qt::WindowModality windowModality);
```

实际上它是从 `QWidget` 类继承过来的一个成员函数，其中 `Qt::WindowModality` 是一个枚举类型，有以下取值。

- ◆ `Qt::NonModal`：非模态。
- ◆ `Qt::WindowModal`：窗口模态。
- ◆ `Qt::ApplicationModal`：应用程序模态。

与模态性相关的成员函数还有以下两个：

```
Qt::WindowModality windowModality() const; // 得到窗口的模态性
bool isModal() const; // 判断窗口是否为模态的
```

20.6.10.3 执行与结果

下面这个函数将以模态方式显示对话框：

```
int exec(); // 这是一个槽
```

同时它也是一个槽。这个函数不管对话框的模态性如何，总是显示模态对话框。调用这个函数的代码将阻塞直到对话框被关闭，返回值表示对话框的结果。注意这个函数总是在其他窗口的代码中调用，一般不在对话框自己的代码中使用。

对话框的结果与下面的函数有关：

```
virtual void done(int r); // 关闭对话框并返回结果 r，这是一个槽
```

这是一个槽，它将使对话框关闭，使对 `exec` 函数的调用返回。参数 `r` 是整型值，但最好使用 `QDialog::DialogCode` 枚举类型所定义的两个值。

- ◆ `QDialog::Accepted`：表示确定。
- ◆ `QDialog::Rejected`：表示取消。

对话框通常有一个确定按钮和一个取消按钮，它们能使对话框关闭并返回相应的值。为了与按钮的 `clicked()` 信号连接，需要用到以下的槽：

```
virtual void accept(); // 槽，等价于 done(QDialog::Accepted)
virtual void reject(); // 槽，等价于 done(QDialog::Rejected)
```

20.6.10.4 打开

下面这个函数将以对窗口模态的方式显示对话框：

```
void open(); // 这是一个槽
```

它也是一个槽。与 `exec` 函数不同的是，`open` 函数将立刻返回而不是等待对话框关闭。它一般用在需要显示模态对话框但又要继续进行工作的场合，比如一个进度对话框。这时候，需要在工

作过程中歇性地调用 `QApplication` 对象的 `processEvents` 成员函数,否则对话框的事件将得不到处理。

20.6.11 QLabel

`QLabel` 类代表标签,它是一个用于显示文本或图像的窗口部件。

20.6.11.1 构造

`QLabel` 类支持以下构造函数:

```
QLabel(QWidget *parent = 0, Qt::WindowFlags f = 0);  
QLabel(const QString &text, QWidget *parent = 0, Qt::WindowFlags f = 0);
```

其中第二个构造函数能够同时通过参数 `text` 给出要显示的文本,因此是最常用的构造方式。

20.6.11.2 属性

`QLabel` 对象的显示内容可以通过属性获取或修改,相关成员函数如下:

```
QString text() const; // 获取显示的文本  
void setText(const QString &text); // 设置显示的文本,这是一个槽  
const QPixmap *pixmap() const; // 获取显示的图像  
void setPixmap(const QPixmap &pixmap); // 设置显示的图像,这是一个槽  
void setNum(int num); // 设置显示的文本为代表整数 num 的字符串,这是一个槽  
void setNum(double num); // 设置显示的文本为代表浮点数 num 的字符串,这是一个槽
```

其中进行设置的函数同时也是槽。新设置的内容将取代原来的内容。

用以下的成员函数则可以清空显示内容:

```
void clear(); // 清空显示内容
```

`alignment` 属性代表显示内容的对齐方式,相关成员函数如下:

```
Qt::Alignment alignment() const; // 获取对齐方式  
void setAlignment(Qt::Alignment align); // 设置对齐方式
```

这里的 `Qt::Alignment` 类型有以下取值。

- ◆ `Qt::AlignLeft`: 水平方向靠左。
- ◆ `Qt::AlignRight`: 水平方向靠右。
- ◆ `Qt::AlignHCenter`: 水平方向居中。
- ◆ `Qt::AlignJustify`: 水平方向调整间距两端对齐。
- ◆ `Qt::AlignTop`: 垂直方向靠上。
- ◆ `Qt::AlignBottom`: 垂直方向靠下。
- ◆ `Qt::AlignVCenter`: 垂直方向居中。
- ◆ `Qt::AlignCenter`: 等价于 `Qt::AlignHCenter` | `Qt::AlignVCenter`。

其中一个水平方向的取值和一个垂直方向的取值可以用“按位或”的方式组合起来以同时指定

两个方向的对齐方式。默认的对齐方式为水平靠左、垂直居中。

indent 属性代表文本的缩进值，相关的成员函数如下：

```
int indent() const; // 获取文本缩进值
void setIndent(int indent); // 设置文本缩进值
```

margin 属性代表显示内容的边距，相关的成员函数如下：

```
int margin() const; // 获取边距
void setMargin(int margin); // 设置边距
```

wordWrap 属性代表显示的文本是否允许换行，相关的成员函数如下：

```
bool wordWrap() const; // 判断是否允许换行
void setWordWrap(bool on); // 设置是否允许换行
```

scaledContents 属性表示显示图像时是否允许缩放，相关的成员函数如下：

```
bool hasScaledContents() const; // 判断是否允许图像缩放
void setScaledContents(bool on); // 设置是否允许图像缩放
```

如果允许缩放，则显示图像时会进行缩放以填满整个显示区域。

20.6.12 QAbstractButton

QAbstractButton 类是 **QPushButton**、**QCheckBox**、**QRadioButton** 等类的基类，它可以代表按钮、单选钮、复选钮等重要窗口部件。通常不会直接使用这个类，这里主要介绍它的一些属性、信号和槽，介绍时暂时将这些窗口部件统称为钮。

20.6.12.1 属性

checkable 属性代表钮是否支持开关状态，比如按钮一般没有开关状态，而单选钮和复选钮都有打开和关闭两个状态。与 **checkable** 属性相关的成员函数如下：

```
bool isCheckedable() const; // 判断是否支持开关状态
void setCheckable(bool on); // 设置是否支持开关状态
```

如果一个钮支持开关状态，则可以用 **checked** 属性设置它的开关状态，相关的成员函数如下：

```
bool isChecked() const; // 判断是否打开
void setChecked(bool on); // 设置是否打开，这是一个槽
```

其中 **setChecked** 函数同时也是一个槽。

autoExclusive 属性表示一个钮是否排他，相关的成员函数如下：

```
bool autoExclusive() const; // 判断是否排他
void setAutoExclusive(bool on); // 设置是否排他
```

排他属性也与钮的开关状态有关，同一个父窗口下的多个排他的钮同时只能有一个是打开的，当一个钮被打开时，其他的钮将自动关闭。



down 属性代表钮是否处于被按下的状态，相关成员函数如下：

```
bool isDown() const; // 判断是否被按下
void setDown(bool on); // 设置是否被按下
```

注意设置这个属性仅仅是改变钮的外观，不会发射任何信号。

通过 **icon** 和 **text** 属性可以设置钮所显示的图标及文本，相关成员函数如下：

```
QIcon icon() const; // 获得图标
void setIcon(const QIcon &icon); // 设置图标
QString text() const; // 获得文本
void setText(const QString &text); // 设置文本
```

shortcut 属性代表钮的快捷键，相关成员函数如下：

```
QKeySequence shortcut() const; // 获得快捷键
void setShortcut(const QKeySequence &key); // 设置快捷键
```

从键盘输入一个钮的快捷键等价于单击这个钮。设置快捷键所需的 **QKeySequence** 对象可用如下方式构造：

```
QKeySequence("Ctrl+P"); // 构造表示 Ctrl+P 组合键的 QKeySequence 对象
```

20.6.12.2 信号

在钮上用鼠标左键或者当钮得到焦点的时候用空格键都能操作这个钮。当钮被按下和松开时，将发射以下信号：

```
void pressed(); // 钮被按下
void released(); // 钮被松开
```

当单击钮时，将发射以下信号：

```
void clicked(bool checked = false);
```

其中参数 **checked** 表示钮是否被打开。

当钮的开关状态变化时，将发射以下信号：

```
void toggled(bool checked);
```

其中参数 **checked** 表示钮是否被打开。

20.6.12.3 槽

下面的两个槽能够模拟钮被单击的过程：

```
void click();
void animateClick(int msec = 100);
```

两者的不同点在于，**click** 函数不会改变钮的外观，它会直接进行单击的处理过程；而 **animateClick** 函数将完全模拟用户按下钮的过程，参数 **msec** 就是钮被按下的毫秒数。两者最终



都会导致 `pressed`, `released` 和 `clicked` 信号的发射。

下面的槽将导致钮的开关状态发生切换：

```
void toggle();
```

当然，它只对有开关状态的钮发生作用。

20.6.13 QPushButton

`QPushButton` 类代表按钮，它继承了 `QAbstractButton` 类，因此也有其全部的属性、信号和槽。

20.6.13.1 构造

`QPushButton` 类支持以下构造函数：

```
QPushButton(QWidget *parent = 0);
QPushButton(const QString &text, QWidget *parent = 0);
QPushButton(const QIcon &icon, const QString &text, QWidget *parent = 0);
```

其中参数 `text` 是按钮上显示的文本，`icon` 是按钮的图标。生成的 `QPushButton` 对象默认是没有开关状态的。

20.6.13.2 属性

按钮的外观一般是有立体感的。通过它的 `flat` 属性可以控制外观是否为扁平状，相关成员函数如下：

```
bool isFlat() const; // 判断外观是否为扁平状
void setFlat(bool flat); // 设置外观是否为扁平状
```

如果这个属性为 `true`，则按钮就是扁平的，否则就是立体的。

如果一个按钮在对话框中，那么它就有默认和自动默认的概念，分别由 `default` 属性以及 `autoDefault` 属性控制，相关的成员函数如下：

```
bool isDefault() const; // 判断按钮是否为默认按钮
void setDefault(bool on); // 设置按钮是否为默认按钮
bool autoDefault() const; // 判断按钮是否为自动默认按钮
void setAutoDefault(bool); // 设置按钮是否为自动默认按钮
```

对话框中同一时刻只能有一个默认按钮，当用户按回车键时，等价于在对话框中单击这个按钮。实际上，对话框中还有一个主默认按钮的概念。一个自动默认按钮得到焦点时，就会变成默认按钮；失去焦点时，主默认按钮就成为默认按钮。而设置一个按钮的默认属性为 `true` 将同时使它成为对话框的主默认按钮。

20.6.14 QCheckBox

`QCheckBox` 类代表复选钮，它继承了 `QAbstractButton` 类，因此也有其全部的属性、信号和槽。

20.6.14.1 构造

QCheckBox 支持以下构造函数：

```
QCheckBox(QWidget *parent = 0);  
QCheckBox(const QString &text, QWidget *parent = 0);
```

其中 **text** 参数表示要显示的文本。生成的复选钮默认是不排他的。

20.6.14.2 属性

tristate 属性用于控制复选钮是否为三态的，相关的成员函数如下：

```
bool isTristate() const; // 判断是否三态  
void setTristate(bool y = true); // 设置是否为三态的
```

三态的复选钮不只有打开和关闭两种状态，其状态由 **checkState** 属性来代表，相关的成员函数如下：

```
Qt::CheckState checkState() const; // 获得状态  
void setCheckState(Qt::CheckState state); // 设置状态
```

这里 **Qt::CheckState** 是一个枚举类型，它有以下取值。

- ◆ **Qt::Unchecked**：未选中。
- ◆ **Qt::PartiallyChecked**：部分选中。
- ◆ **Qt::Checked**：选中。

部分选中的状态经常用来表示一个选项树中某个选项的子选项有的选中、有的未选的状态。如果一个复选钮不是三态的，则也可以用从 **QAbstractButton** 继承来的 **checked** 属性操作它的状态。

20.6.14.3 信号

当复选钮的状态发生变化时，将发射以下信号：

```
void stateChanged(int state);
```

其中参数 **state** 表示复选钮的新状态。

20.6.15 QRadioButton

QRadioButton 类表示单选钮，它继承了 **QAbstractButton** 类，因此也有其全部的属性、信号和槽。**QRadioButton** 类支持以下构造函数：

```
QRadioButton(QWidget *parent = 0);  
QRadioButton(const QString &text, QWidget *parent = 0);
```

其中参数 **text** 表示要显示的文本。生成的单选钮默认是排他的。

20.6.16 QLineEdit

QLineEdit 类表示编辑框，它可以让用户输入一个单行文本。

20.6.16.1 构造

QLineEdit 类支持以下构造函数：

```
QLineEdit(QWidget *parent = 0);
QLineEdit(const QString &contents, QWidget *parent = 0);
```

其中 **contents** 表示编辑框中显示的内容。

20.6.16.2 属性

alignment 属性表示显示文本的对齐方式，相关成员函数如下：

```
Qt::Alignment alignment() const; // 获取对齐方式
void setAlignment(Qt::Alignment flag); // 设置对齐方式
```

它的含义与 **QLabel** 类的 **alignment** 属性相同。

maxLength 属性表示编辑框可以容许的最大输入长度，相关成员函数如下：

```
int maxLength() const; // 获取最大输入长度
void setMaxLength(int len); // 设置最大输入长度
```

readOnly 属性表示编辑框的内容是否为只读的，即内容是否可以被用户修改，相关成员函数如下：

```
bool isReadOnly() const; // 判断是否为只读的
void setReadOnly(bool on); // 设置是否为只读的
```

text 属性表示编辑框的内容，相关成员函数如下：

```
QString text() const; // 获取编辑框的内容
void setText(const QString &text); // 设置编辑框的内容，这是一个槽
```

其中 **setText** 函数同时也是一个槽。要注意编辑框的内容不一定是编辑框显示的内容，比如一个密码输入编辑框，用户输入的密码是不显示出来的。如果要得到编辑框显示的内容，则需使用下面的成员函数：

```
QString displayText() const; // 得到显示的内容
```

frame 属性控制编辑框有没有边框，相关成员函数如下：

```
bool hasFrame() const; // 判断有没有边框
void setFrame(bool on); // 设置有没有边框
```

下面两个成员函数虽然不是属性，但也与文本的显示有关：

```
void setTextMargins(int left, int top,
```

```
int right, int bottom); // 设置边距
void getTextMargins(int *left, int *top,
int *right, int *bottom) const; // 获取边距
```

这两个函数分别用于设置和获取文本显示的边距，上、下、左、右 4 个边距可以独立进行设置。

20.6.16.3 信号

当编辑框的内容发生变化时，将发射以下信号：

```
void textChanged(const QString &text);
```

其中参数 `text` 是变化后的内容。

当编辑框的内容被编辑时，将发射以下信号：

```
void textEdited(const QString &text);
```

其中参数 `text` 是编辑后的内容。它与 `textChanged` 信号的主要区别在于，它只在用户进行修改的时候发射，在程序中用 `setText` 修改则不发射，而后者则在两种情况下都会发射。

当在编辑框中按下回车键时，将发射以下信号：

```
void returnPressed();
```

这可以作为用户结束输入的一种条件。或者使用以下这个信号：

```
void editingFinished();
```

这个信号将在按下回车键或者编辑框失去焦点时发射。

20.6.16.4 槽

调用下面的槽可以清空编辑框中的内容：

```
void clear();
```

20.7 Qt 综合应用

这一节将首先以一个完整的应用程序例子来说明对各种 Qt 类的综合运用，然后说明如何进行 Qt 的国际化编程。

20.7.1 软件设计

这个应用程序要实现的功能是一个猜数字的小游戏。游戏的规则很简单，首先由计算机随机产生 4 个范围在 0~9 内的数字但是不显示给玩家，由玩家进行试猜。当玩家输入 4 个数字后，计算机进行统计，给出这 4 个数字与答案中 4 个数字相同的个数及位置也相同的个数。玩家根据这个结果进行下一次的试猜，直到所输入的 4 个数字及数字的位置与答案完全相同为成功，或者达到规定的次数为失败。

20.7.1.1 产生随机数

产生随机数可以使用系统的 `rand()` 函数，为了使每次运行程序时的随机数序列不同，可以在程序启动时重设随机数种子，如：

```
srand(time(NULL)); /* 重设随机数种子为当前时间 */
```

这里不需要 Qt 的支持。

20.7.1.2 输入数字

输入数字可以使用 `QLineEdit` 类。为了增加游戏的可玩性，对于 4 个数字可以分别用 4 个编辑框接受输入。当一个编辑框内输入完毕后，焦点自动移到下一个未输入的编辑框内；当 4 个编辑框全部有输入后，自动开始统计相同数字及相同位置的个数。

为了让用户可以根据以前的试猜结果进行新的尝试，所有已经试猜的数字和统计结果都应该显示在屏幕上，但不可以修改，这可以用 `QLineEdit` 类的只读属性来控制。这样一来，每进行一次新的试猜，就要重新启用 4 个编辑框，总的个数将是 4 乘以允许的试猜次数。这个数目比较大，因此不适用于 `designer` 进行界面设计，只能自己书写代码来构造界面。编辑框对象指针可以定义为二维数组，编辑框在界面上的排列可以用 `QGridLayout` 类来控制。

还有一个问题是控制用户的输入只能是 0~9 的数字。首先，可以设置 `QLineEdit` 类的 `maxLength` 属性为 1 以控制输入长度。但这并不能阻止用户输入数字以外的字符，虽然可以通过判断输入结果并提示用户重新输入的方法来解决，但将大大影响游戏的流畅性。实际上，`QLineEdit` 对象支持设置校验器的方式来控制输入，所用的成员函数如下：

```
void setValidator(const QValidator *v);
```

这里的 `QValidator` 类代表校验器，这是一个抽象类，必须继承它并实现它的纯虚函数方可使用。好在 Qt 已经实现了若干个校验器，其中的 `QIntValidator` 类专门用于校验输入是否为整数并且可以限定整数的范围，它的常用构造函数如下：

```
QIntValidator(int minimum, int maximum, QObject *parent);
```

其各个参数的含义解释如下。

- ◆ `minimum`：允许的最小值。
- ◆ `maximum`：允许的最大值。
- ◆ `parent`：指向父对象。

`QLineEdit` 对象设置了校验器以后，每次用户进行编辑时都会经过校验器内函数的检验，如果不符合要求，则新内容无效，编辑框的内容仍保持旧内容。

为了使程序简单，可以将接受输入的 4 个编辑框的“已编辑”信号连接到同一个槽上进行处理。这样就必须在槽里分辨是哪一個编辑框发射的信号。用 `QObject` 类的以下成员函数可以达到这个目的：

```
QObject *sender() const;
```

它可以得到信号的发送者的指针，将这个指针与 4 个编辑框的指针一一比较即可确定是哪一个编辑框在发射信号。

必须注意，这个函数在某些情况（比如信号是异步发送的、发送者已销毁）下并不能得到有效的指针，因此要慎用。

20.7.1.3 显示信息

显示信息可以使用 `QLabel` 类，显示统计结果的标签与用于输入的编辑框是对应的，可以放在同一个栅格布局中。

20.7.1.4 开始及中途放弃

游戏开始及中途放弃可以使用 `QPushButton` 类控制。因为只有开始后才能中途放弃，因此可以只用一个按钮。初始化时，按钮的提示文本是“开始”，它的 `clicked()` 信号连接在用于开始游戏的槽上。当用户单击“开始”按钮后，游戏开始，然后将按钮的提示文本变为“显示答案”，并将 `clicked()` 信号改连在用于显示答案的槽上。当用户单击“显示答案”按钮后，显示答案并结束游戏，按钮回到初始的状态。这里，信号与槽的连接可以动态更改，体现了其灵活性。

20.7.1.5 结果提示

当用户成功或失败后应有相应的提示，最简单的办法就是弹出一个对话框。这个对话框可以自己设计。这里为了简单，采用了系统的 `QMessageBox` 类来显示对话框。它可以显示默认的对话框，其常用的构造函数如下：

```
QMessageBox(Icon icon, const QString &title, const QString &text,  
            StandardButtons buttons = NoButton, QWidget *parent = 0,  
            Qt::WindowFlags f = Qt::Dialog | Qt::MSWindowsFixedSizeDialogHint);
```

其各个参数的含义解释如下。

- ◆ `icon`: 表示对话框的提示图标。
- ◆ `title`: 对话框标题。
- ◆ `text`: 对话框中显示的文本。
- ◆ `buttons`: 对话框的标准按钮。
- ◆ `parent`: 对话框的父窗口。
- ◆ `f`: 构造窗口的标志。

这里的 `Icon` 类型是 `QMessageBox` 的内嵌类型，有以下取值。

- ◆ `QMessageBox::NoIcon`: 无图标。
- ◆ `QMessageBox::Question`: 问号图标。
- ◆ `QMessageBox::Information`: 信息图标。
- ◆ `QMessageBox::Warning`: 警告图标。
- ◆ `QMessageBox::Critical`: 严重错误图标。

StandardButtons 类型也是 QMessageBox 的内嵌类型，有以下取值。

- ◆ QMessageBox::Ok: 确定。
- ◆ QMessageBox::Open: 打开。
- ◆ QMessageBox::Save: 保存。
- ◆ QMessageBox::Cancel: 取消。
- ◆ QMessageBox::Close: 关闭。
- ◆ QMessageBox::Discard: 不保存关闭。
- ◆ QMessageBox::Apply: 应用。
- ◆ QMessageBox::Reset: 重置。
- ◆ QMessageBox::RestoreDefaults: 重置默认值。
- ◆ QMessageBox::Help: 帮助。
- ◆ QMessageBox::SaveAll: 全部保存。
- ◆ QMessageBox::Yes: 是。
- ◆ QMessageBox::YesToAll: 全部是。
- ◆ QMessageBox::No: 否。
- ◆ QMessageBox::NoToAll: 全部否。
- ◆ QMessageBox::Abort: 放弃。
- ◆ QMessageBox::Retry: 重试。
- ◆ QMessageBox::Ignore: 忽略。
- ◆ QMessageBox::NoButton: 无按钮。

这些值可以用“按位或”的方式组合起来，表示同时有几个按钮。

QMessageBox 类覆盖了 QDialog 类的 exec 方法，将这个函数的返回值变为上述表示按钮的值，以表示是由于这个按钮的动作导致对话框退出的。

20.7.1.6 功能扩展

生成随机数时，可以考虑加以控制，使之不能出现相同数字。将能不能出现相同数字作为一个选项，程序的功能就得到了扩展。这里采用两个单选钮的形式来控制这个选项。

20.7.2 源码实现

应用程序由三个源文件组成：main.cpp，wdg_guess.h 和 wdg_guess.cpp。

main.cpp 是程序的主模块，它的源码如下：

```
// 文件名: main.cpp
// 说明: 猜数字游戏主模块

#include <QApplication>
#include "wdg_guess.h"

int main(int argc, char *argv[])
{
```



```

    QApplication app(argc, argv);
    WdgGuess wdg;
    wdg.show();
    return app.exec();
}

```

可以看出主模块的代码仍然十分简单。

`wdg_guess.h` 文件是窗口类 `WdgGuess` 的定义，源码如下：

```

// 文件名: wdg_guess.h
// 说明: 猜数字游戏 WdgGuess 类定义

#ifndef WDG_GUESS_INCLUDED
#define WDG_GUESS_INCLUDED

#include <QWidget>

// 以下为类的前向声明
class QHBoxLayout;
class QVBoxLayout;
class QGridLayout;
class QLabel;
class QPushButton;
class QRadioButton;
class QLineEdit;

class WdgGuess: public QWidget {
    Q_OBJECT
public:
    WdgGuess();
    virtual ~WdgGuess();
protected:
    static const int nrNum = 4; // 数字的个数
    static const int nrTimes = 10; // 允许尝试的次数
    int num[nrNum]; // 生成的数字
    int times; // 当前尝试的次数
    QVBoxLayout *layout; // 默认布局
    QHBoxLayout *layoutRadio; // 放置两个单选钮
    QGridLayout *layoutGrid; // 放置编辑框和标签阵列
    QLabel *labelNum[nrNum]; // 显示生成的数字
    QLabel *labelRightNum; // 显示正确数字的个数
    QLabel *labelRightPlace; // 显示正确位置的个数
    QLabel *labelResult[nrTimes][2]; // 显示统计结果
    QPushButton *button; // 开始与显示结果
    QRadioButton *radioSame; // 允许相同数字选项
    QRadioButton *radioNoSame; // 禁止相同数字选项
    QLineEdit *editNum[nrTimes][nrNum]; // 编辑框阵列
protected:
    void clearPad(); // 清理编辑框及统计结果
    void disableAllEdit(); // 禁用所有编辑框
    void beginThisTime(); // 开始本次尝试

```



```

    void endThisTime(); // 结束本次尝试
    bool showStat(); // 显示统计结果
    void victory(); // 成功
    void failed(); // 失败
protected slots:
    void begin(); // 开始
    void showAnswer(); // 显示答案
    void processInput(const QString &text); // 处理用户输入
};

#endif

```

WdgGuess 类的实现则在 wdg_guess.cpp 文件中, 源码如下:

```

// 文件名: wdg_guess.cpp
// 说明: 猜数字游戏 WdgGuess 类实现

#include "wdg_guess.h"
#include <QVBoxLayout>
#include <QGridLayout>
#include <QLabel>
#include <QPushButton>
#include <QRadioButton>
#include <QLineEdit>
#include <QMessageBox>
#include <QIntValidator>

WdgGuess::WdgGuess()
{
    int i, j;
    setWindowTitle(tr("Guess Number")); // 设置窗口标题
    // 生成布局、单选钮及按钮
    layout = new QVBoxLayout(this);
    layoutRadio = new QHBoxLayout;
    layout->addLayout(layoutRadio);
    radioSame = new QRadioButton(tr("Allow Sameness"));
    layoutRadio->addWidget(radioSame);
    radioNoSame = new QRadioButton(tr("No Sameness"));
    layoutRadio->addWidget(radioNoSame);
    button = new QPushButton(tr("Begin"));
    layout->addWidget(button);
    layoutGrid = new QGridLayout;
    layout->addLayout(layoutGrid);
    // 生成用于显示数字的标签
    for (i = 0; i < nrNum; i++) {
        labelNum[i] = new QLabel("0"); // 初始状态, 显示 0
        layoutGrid->addWidget(labelNum[i], 0, i);
    }
    // 生成校验器, 只允许输入 0~9 的数字
    QValidator *validator = new QIntValidator(0, 9, this);
    // 生成并放置编辑框

```

```

    for (i = 0; i < nrTimes; i++) {
        for (j = 0; j < nrNum; j++) {
            editNum[i][j] = new QLineEdit;
            editNum[i][j]->setMaxLength(1); // 最大输入长度为 1
            editNum[i][j]->setValidator(validator); // 安装校验器
            layoutGrid->addWidget(editNum[i][j], i+1, j);
        }
    }
    // 生成正确数字及正确位置的两个标签
    labelRightNum = new QLabel(tr("Right Num"));
    layoutGrid->addWidget(labelRightNum, 0, nrNum);
    labelRightPlace = new QLabel(tr("Right Place"));
    layoutGrid->addWidget(labelRightPlace, 0, nrNum+1);
    // 生成用于显示结果的标签
    for (i = 0; i < nrTimes; i++) {
        for (j = 0; j < 2; j++) {
            labelResult[i][j] = new QLabel;
            layoutGrid->addWidget(labelResult[i][j], i+1, nrNum+j);
        }
    }
    // 将按钮的单击信号连接到开始槽
    connect(button, SIGNAL(clicked()), this, SLOT(begin()));
    // 重设随机数种子
    srand(time(NULL));
    // 禁用所有编辑框
    disableAllEdit();
    // 设置单选钮
    radioNoSame->setChecked(true);
}

WdgGuess::~WdgGuess()
{
}

// 清理编辑框及显示结果
void WdgGuess::clearPad()
{
    int i, j;
    for (i = 0; i < nrTimes; i++) {
        for (j = 0; j < nrNum; j++) {
            editNum[i][j]->setReadOnly(false); // 取消只读属性
            editNum[i][j]->clear();
        }
        labelResult[i][0]->clear();
        labelResult[i][1]->clear();
    }
}

// 禁用所有编辑框
void WdgGuess::disableAllEdit()
{
}

```

```

    int i, j;
    for (i = 0; i < nrTimes; i++) {
        for (j = 0; j < nrNum; j++) {
            editNum[i][j]->setDisabled(true);
        }
    }
}

// 开始本次尝试
void WdgGuess::beginThisTime()
{
    int i;
    for (i = 0; i < nrNum; i++) {
        // 将本行编辑框的已编辑信号连接到输入处理槽
        connect(editNum[times][i], SIGNAL(textEdited(const QString &)),
                this, SLOT(processInput(const QString &)));
        // 使能这些编辑框
        editNum[times][i]->setEnabled(true);
    }
    // 将焦点放在本行的第一个编辑框上
    editNum[times][0]->setFocus();
}

// 结束本次尝试
void WdgGuess::endThisTime()
{
    int i;
    for (i = 0; i < nrNum; i++) {
        // 将本行编辑框的已编辑信号的连接断开
        disconnect(editNum[times][i], SIGNAL(textEdited(const QString &)),
                this, SLOT(processInput(const QString &)));
        // 将编辑框设为只读, 不可以再修改
        editNum[times][i]->setReadOnly(true);
    }
}

// 显示统计结果, 返回 true 表示已成功, false 表示未成功
bool WdgGuess::showStat()
{
    int i, j;
    int m = 0; // 用于统计正确数字的个数
    int n = 0; // 用于统计正确位置的个数
    for (i = 0; i < nrNum; i++) {
        for (j = 0; j < nrNum; j++) {
            if (editNum[times][i]->text().toInt() == num[j]) {
                m++;
                if (i == j) n++;
                break;
            }
        }
    }
}

```



```

    // 显示结果
    labelResult[times][0]->setNum(m);
    labelResult[times][1]->setNum(n);
    // 正确位置达到数字个数说明成功, 返回 true, 否则返回 false
    return (n == nrNum) ? true : false;
}

// 成功
void WdgGuess::victory()
{
    QMessageBox msg(QMessageBox::Question,
        tr("Success"), tr("Good job! Another game?"),
        QMessageBox::Ok | QMessageBox::Cancel, this);
    showAnswer(); // 显示答案
    // 显示对话框, 如果用户输入 Ok 则开启新的一局
    if (msg.exec() == QMessageBox::Ok) button->click();
}

// 失败
void WdgGuess::failed()
{
    QMessageBox msg(QMessageBox::Critical,
        tr("Fail"), tr("Game over!"),
        QMessageBox::Ok, this);
    showAnswer(); // 显示答案
    msg.exec(); // 显示对话框
}

// 开始槽
void WdgGuess::begin()
{
    int i, j;
    times = 0; // 尝试次数清零
    for (i = 0; i < nrNum; i++) {
        if (radioNoSame->isChecked()) { // 如果有禁止相同数字选项
            do { // 反复生成随机数, 直到所有数字不同为止
                num[i] = rand()%10;
                for (j = 0; j < i && num[j] != num[i]; j++) {}
            } while (j < i);
        } else { // 否则允许相同数字
            num[i] = rand()%10; // 直接生成随机数
        }
        labelNum[i]->setText("*"); // 数字显示为 * 号
    }
    // 修改按钮文本为显示答案
    button->setText(tr("Show Answer"));
    // 断开按钮的单击信号与开始槽的连接
    disconnect(button, SIGNAL(clicked()), this, SLOT(begin()));
    // 将按钮的单击信号重新连接到显示答案槽
    connect(button, SIGNAL(clicked()), this, SLOT(showAnswer()));
    // 清理编辑框及显示结果

```




```

    clearPad();
    // 开始第一次尝试
    beginThisTime();
}

// 显示答案槽
void WdgGuess::showAnswer()
{
    int i;
    for (i = 0; i < nrNum; i++) labelNum[i]->setNum(num[i]); // 显示数字
    disableAllEdit(); // 禁用所有编辑框
    // 修改按钮标题为开始
    button->setText(tr("Begin"));
    // 断开按钮的单击信号与显示答案槽的连接
    disconnect(button, SIGNAL(clicked()), this, SLOT(showAnswer()));
    // 将按钮的单击信号重新连接到开始槽
    connect(button, SIGNAL(clicked()), this, SLOT(begin()));
}

// 处理输入槽
void WdgGuess::processInput(const QString &text)
{
    int i;
    Q_UNUSED(text); // 未使用的参数, 避免产生编译警告
    QLineEdit *edit = (QLineEdit *)sender(); // 得到信号的发送者
    if (!edit) return;
    // 查找信号的发送者是哪一个编辑框
    for (i = 0; i < nrNum && edit == editNum[times][i]; i++) {}
    if (i == nrNum) return; // 未找到, 返回
    // 查找本行内有没有未输入的编辑框
    for (i = 0; i < nrNum && !editNum[times][i]->text().isEmpty(); i++) {}
    if (i == nrNum) { // 没有未输入的编辑框, 说明本次尝试已结束
        endThisTime(); // 结束本次尝试
        if (showStat()) { // 显示统计结果, 如果成功
            victory(); // 成功
            return;
        }
        times++; // 尝试次数增加
        if (times == nrTimes) { // 如果尝试次数达到限定次数, 说明失败
            failed(); // 失败
            return;
        } else {
            beginThisTime(); // 开始本次尝试 (尝试次数已增加)
        }
    } else { // 还有未输入的编辑框
        editNum[times][i]->setFocus(); // 将焦点放在未输入的编辑框上
    }
}
}

```



20.7.3 运行结果

例程运行以后的界面如图 20.15 所示。

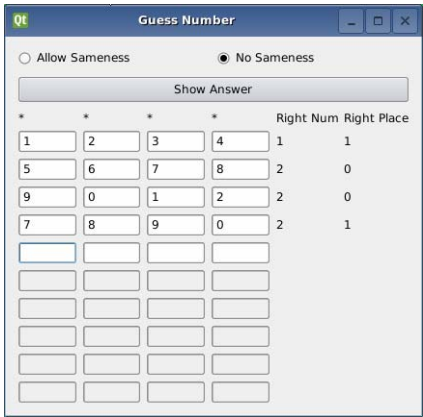


图 20.15 猜数字游戏运行结果

20.7.4 Qt 国际化编程

一般来说，编程时应尽量避免在代码中直接使用汉字。因为汉字有各种编码格式的问题，在不同的开发环境中，有可能显示为乱码甚至导致无法编译。

Qt 采用 Linux 系统上通行的翻译文件的方式来解决应用程序的国际化问题，做法如下。

- ◆ 书写程序时，代码中嵌入的字符串可以只用英文甚至代号来书写。
- ◆ 随应用程序提供一个翻译文件，它给出了源代码中字符串到目标语言字符串的对应关系。
- ◆ 程序运行时将根据源代码中的字符串去翻译文件中查找对应的目标语言字符串，最终显示给用户的是目标语言。

这样，更换不同的翻译文件，同一个程序就能显示不同的语言，不需要重新编译。

这里将以前述猜数字游戏应用程序为例来说明 Qt 4.5.2 中国际化编程的基本过程。

20.7.4.1 提取字符串

要想得到翻译文件，必须首先将源码中待翻译的字符串提取出来。这可以利用 Qt 提供的工具 `lupdate` 来做。首先要在工程文件中加入以下一行：

```
TRANSLATIONS += guess_zh_CN.ts
```

这一行表示要产生翻译文件 `guess_zh_CN.ts`，然后运行 `lupdate`：

```
lupdate guess.pro # guess.pro 为工程文件名
```

这样就会产生一个名为 `guess_zh_CN.ts` 的文件，它是用于翻译的基础。

为了让 `lupdate` 能够找到，源码中待翻译的字符串必须做特殊的处理。一般来说，在 Qt 对象类中的字符串可以用 `tr` 函数包围起来，如：



```
button->setText(tr("Begin"));
```

这里 `tr` 实际上是 Qt 对象类的一个成员函数，它由 `moc` 工具自动生成，所以要使用 `tr` 函数的类定义时必须有 `Q_OBJECT`。

如果字符串是放在初始化数据中的，则可以用 `QT_TR_NOOP` 宏包围起来，如：

```
char *str = QT_TR_NOOP("Begin");
button->setText(tr(str)); // 注意这里仍然要加 tr
```

对于编译器来说，这个宏就等于直接写它里面的字符串，但是它可以被 `lupdate` 识别出来。

如果待翻译的字符串不在 Qt 对象类里面，则可以用 `QApplication` 类的 `translate` 函数包围起来，如：

```
qDebug() << qApp->translate("MyClass", "Debug Info");
```

这里，`translate` 函数的第二个参数才是要翻译的字符串，第一个参数表示上下文。不同上下文内的相同字符串可以翻译成不同的结果。实际上，`tr` 函数也是根据上下文进行翻译的，它用的上下文就是类的名称。

对应于 `translate` 函数的初始化数据则需要用 `QT_TRANSLATE_NOOP` 包围起来。

如果一个字符串只是用于程序内部使用，不显示给用户，则没有必要翻译，比如对象的名称。

20.7.4.2 翻译

由 `lupdate` 得到的字符串放在一个 XML 格式的文件中，因此可以直接用文本编辑器进行修改。更为方便的是使用 Qt 4.5.2 提供的工具 `linguist`，它是一个图形化的翻译文件编辑工具。在终端上输入以下命令即可启动 `linguist`：

```
linguist
```

`linguist` 的界面如图 20.16 所示。

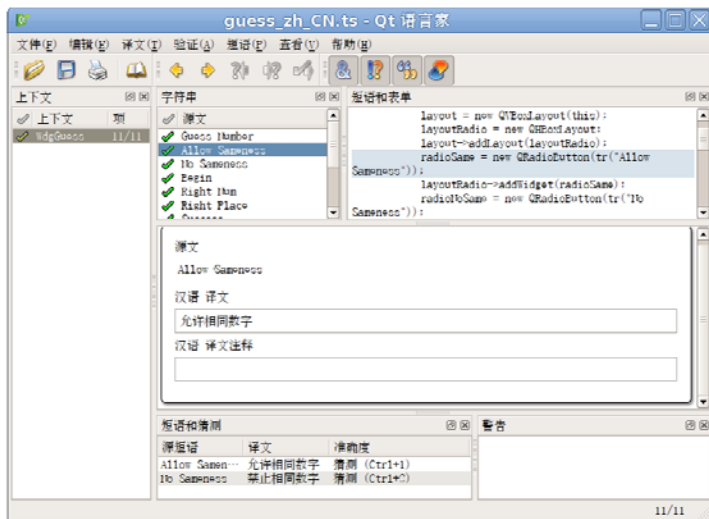


图 20.16 linguist 的界面

在 **linguist** 里，我们可以方便地输入翻译的结果，输入完毕之后按 **Ctrl+Enter** 组合键确认或者使用工具栏确认。编辑的结果可以随时存盘。

如果再次运行 **lupdate**，则已有的翻译结果会被保留，只是增加程序中新出现的待翻译字符串。

20.7.4.3 转换

ts 格式的翻译文件并不能直接被 **Qt** 程序使用，需要转换成所要求的 **qm** 格式的文件才行。为此，**Qt 4.5.2** 提供了一个工具 **lrelease**，用法如下：

```
lrelease guess.pro
```

这行命令将工程中的所有翻译文件全部转化为 **qm** 格式。

20.7.4.4 加载翻译器

在源代码中，需要向 **QApplication** 对象加载翻译器。例如，上述猜数字游戏的主模块将修改为以下内容：

```
// 文件名: main.cpp
// 说明: 猜数字游戏主模块

#include <QApplication>
#include <QTranslator>
#include <QLocale>
#include <QLibraryInfo>
#include "wdg_guess.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    // 设置字体为 unifont
    app.setFont(QFont("unifont", 16, 5));
    QTranslator qtTrans;
    // 给翻译器加载翻译文件，这个是 Qt 自己的翻译文件
    qtTrans.load("qt_"+QLocale::system().name(),
                QLibraryInfo::location(QLibraryInfo::TranslationsPath));
    app.installTranslator(&qtTrans); // 加载翻译器
    QTranslator myTrans;
    // 给翻译器加载翻译文件，这个是应用程序自己的翻译文件
    myTrans.load("guess_zh_CN", ".");
    app.installTranslator(&myTrans); // 加载翻译器
    WdgGuess wdg;
    wdg.show();
    return app.exec();
}
```

翻译以后程序的运行结果如图 20.17 所示。



图 20.17 猜数字游戏汉化之后的界面

如果程序运行时找不到需要的翻译文件，或者文件中不包含要翻译的字符串，那么就会原样显示翻译前的字符串。



第 21 章 嵌入式数据库编程

由于嵌入式系统的软硬件环境的限制，一般的大型数据库无法使用，所以需要结合嵌入式系统的特点，选择合适的数据库平台来完成嵌入式数据库系统的构建。一般来说，嵌入式系统上应用的数据库平台应有如下的特点。

一、体积小巧

嵌入式系统对数据库平台自身的体积、数据的存储与程序运行所占的内存一般都有较强的空间限制。

二、功能完备

嵌入式开发中有很多应用需要有一个功能较为完备的数据库平台来实现对数据的管理。对开发人员来说，要求采用的数据库技术提供完备的开发文档而且易于开发。

三、源码开放

对于产品的开发，开放源代码不仅可以减少产品的成本，更重要的是为产品的维护和升级提供了最为彻底的解决手段。

现在广为流行的 SQLite 嵌入式数据库就具有了上述的特点，本章的主要内容就是讲解 SQLite 数据库编程技术。

21.1 基本 SQL 语句

这一节将主要介绍一些基本的 SQL 语句的用法。作为一个轻量级的数据库开发平台，SQLite 并不支持 SQL 语言的全部语法，而且对于嵌入式系统来说，这些基本的 SQL 语句已经可以满足大部分数据库的应用。

21.1.1 数据库与表

SQL 数据库是关系数据库的一种，数据库中的数据被组织成表的形式。表可以形象地理解为一张表格，它由若干拥有相同字段的记录组成。当然，表也可以是空的，即拥有 0 条记录。字段指的是记录中的数据域，它有不同的类型，可以是一个数值，也可以是字符串。

每个表中可以指定一个或多个字段为主键，表中所有记录的主键的值都不能重复。这种不重复是在插入记录时控制的。如果向一个表插入记录时，这个表中已有与插入记录具有相同主键的记录，则插入操作将失败。

SQLite 将每个数据库都保存成一个文件。如图 21.1 所示是一个数据库，数据库中有两张表，名称分别为 Student 和 Score。表 Student 用来保存学生的基本信息，如学号 (ID)，姓名 (Name)、性别 (Sex)、年龄 (Age)；而表 Score 用来保存成绩，包含 ID、Subject 和 Score 三个字段，

分别表示学号、课程和对应的分数。注意在表 **Student** 中，学生的学号是各不相同的，因此可以作为主键；而在表 **Score** 中，一个学生有多个课程的成绩，因此 **ID** 字段不能再单独作为主键。

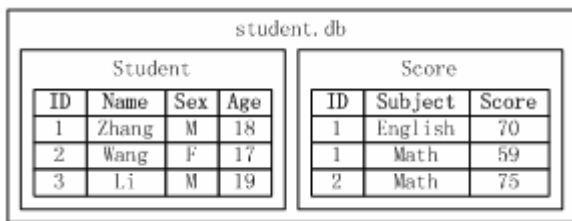


图 21.1 数据库示意

在以下对 SQL 语句用法的介绍中均使用这个数据库作为例子。

21.1.2 创建和删除表

用 **create** 语句可以在数据库中创建表：

```
create table student (
    ID    int(5),
    name  char(10),
    sex   char(1),
    age   int(3),
    primary key(ID)
);
```

这里 **create**, **table**, **int**, **char**, **primary key** 都是 SQL 语言的关键字。创建表时可以指定每个字段的类型和所能存储数据的最大宽度，也可以指定一个或多个主键。这里创建了一个名称为 **student** 的新表，包含 4 个字段：**ID**, **name**, **sex**, **age**。其中 **ID** 字段是最大宽度为 5 的整数，并且被指定为主键，**name** 字段则是最大宽度为 10 的字符串。



在 SQL 语言中，无论是关键字还是表名、字段名都不区分大小写，但记录的值是区分大小写的。SQL 语言并不要求语句的末尾有分号，但在 SQLite 中，分号可以作为语句的结束符来使用。

用 **drop** 语句可以从数据库中删除指定的表：

```
drop table student;
```

这里的 **drop** 也是 SQL 语言的关键字，**student** 是被删除的表。表被删除以后，表中所存储的数据将全部丢弃。

21.1.3 插入、修改及删除记录

用 **insert** 语句可以向表中插入一条新记录：

```
insert into student values(1, 'Zhang', 'M', 18);
```

这里的 **insert**, **into** 和 **values** 都是 SQL 语言的关键字，**student** 则是被插入记录的表的名称。

称。**values** 关键字后面的圆括号中所列的值就是新记录的各个字段对应的值，注意字符串需要用单引号包围起来。

用 **update** 语句可以修改表中的记录：

```
update student set Sex = 'F', Age = 20 where Name = 'Zhang';
```

这里的 **update**, **set** 及 **where** 都是 SQL 语言的关键字。**where** 关键字及其后的部分表示一个过滤条件，如果省略掉 **where** 关键字和它后面的部分，则语句的作用就是修改表 **student** 中的所有记录，将它们 **Sex** 字段的值设为 **F**，**Age** 字段的值设为 **20**。有了这里的 **where** 子句，则只有 **Name** 字段为 **Zhang** 的记录会被修改。**where** 作为指定过滤条件的手段在很多 SQL 语句里都扮演着重要的角色。

用 **delete** 语句可以删除表中的记录：

```
delete from student where ID = 2;
```

这里的 **delete** 和 **from** 都是 SQL 语言的关键字。同样，如果没有 **where** 子句，则表 **student** 中的所有记录均被删除，成为空表。有了这条 **where** 子句的过滤作用，则只有 **ID** 字段为 **2** 的记录被删除。如果 **ID** 字段还是主键，那么最多只有一条记录被删除。

21.1.4 条件表达式

在使用 **where** 子句时，需要一个条件表达式，满足这个条件的记录将被操作。可以用比较操作符将两个值连接起来构成一个条件表达式，SQL 语言中常用的一些比较操作符如表 21.1 所示。除了 **like** 之外，这些操作符既可以比较数值，也可以比较字符串。

表 21.1 SQL 语言中的比较操作符

操作符	功能
=	判断是否等于，例如 ID = 2, Name = 'Zhang'
<>	判断是否不等于，例如 ID <> 2, Sex <> 'M'
<	判断是否小于，例如 Age < 100
>	判断是否大于
<=	判断是否不大于
>=	判断是否不小于
like	模式匹配，例如 Name like 'Zh%', 表示如果 Name 的值开头为 Zh 则判断为真

几个表达式还可以用逻辑操作符组合起来形成更复杂的表达式。SQL 语言中的逻辑操作符如表 21.2 所示。

表 21.2 SQL 语言中的逻辑操作符

操作符	含义
AND	逻辑与，例如 Name like 'Zh%' AND Age < 30
OR	逻辑或
NOT	逻辑非，例如 NOT (Sex = 'F')

21.1.5 数据库查询

查询是对数据库的最常用操作。查询的结果可以理解为得到了一个新表,当然这个表并没有保存在数据库里,而是以某种方式显示给查询者。

21.1.5.1 单表查询

用 `select` 语句可以对数据库进行查询,基本的用法如下:

```
select ID, Name from student where Sex = 'F';
```

这里 `select` 关键字后面的 `ID` 和 `Name` 是结果中希望出现的字段, `student` 是被查询的表的名称, `where` 子句仍然用来指定过滤条件,符合条件的记录将出现在结果中。

如果希望结果中出现表的所有字段,则可以用 `*` 代表所有字段名,如:

```
select * from score where score < 60;
```

如果希望结果中没有重复的记录,则可以使用 `distinct` 关键字,如:

```
select distinct * from score;
```

如果要将结果按某个字段的值进行排序,可以使用 `order by` 子句,如:

```
select * from score order by score asc;
select * from score order by score desc;
```

这里 `order by` 后面的字段名 `score` 是排序时被比较的字段,也可以有多个字段同时被比较。`asc` 关键字表示按升序排列, `desc` 关键字表示按降序排列,如果省略,则默认按升序排列。

21.1.5.2 多表联合查询

查询数据库时,也可以将多个表的内容放在一起进行查询,比如:

```
select student.name from student, score
where student.id = score.id and score.score < 60;
```

在这条语句中, `from` 关键字后面出现了两个表名 `student` 和 `score`,表示对这两个表进行联合查询。查询结果中需要的字段是表 `student` 中的 `name` 字段,而过滤的条件则是两个表中的 `id` 字段相等,且表 `score` 中的 `score` 字段的值小于 60。实际作用就是显示不及格的学生的姓名。由于表示姓名和分数的两个字段分别在两个表里,所以必须进行双表联合查询。

进行多表联合查询时需要注意,在指定字段时必须在前面加上字段所在的表名以使语义明确。可以使用表的别名以避免多次输入较长的表名,并且使代码更清晰,如:

```
select A.name from student A, score B where A.id = B.id and B.score < 60;
```

在这条语句中, `from` 关键字后的表名都指定了一个别名,这样在语句的其他部分中就可以使用这些别名了,注意别名只在本语句中有效。

多表联合查询可视为在一张由多个表的记录两两组合而成的大表中进行查询。如果对如图

21.1 所示的数据库中的表 **Student** 和表 **Score** 进行联合查询，则组合后的大表的内容如表 21.3 所示。

表 21.3 双表内容组合

Student.ID	Student.Name	Student.Sex	Student.Age	Score.ID	Score.Subject	Score.Score
1	Zhang	M	18	1	English	70
1	Zhang	M	18	1	Math	59
1	Zhang	M	18	2	Math	75
2	Wang	F	17	1	English	70
2	Wang	F	17	1	Math	59
2	Wang	F	17	2	Math	75
3	Li	M	19	1	English	70
3	Li	M	19	1	Math	59
3	Li	M	19	2	Math	75

可见，组合后的大表的记录条数等于各个表的记录条数之积。因此多表联合查询的效率比较低，并且语句的不同写法也可能对查询效率有显著的影响。

21.2 建立 SQLite3 开发平台

在 Debian 5.0 系统上，可直接用如下命令安装 **sqlite3**：

```
sudo apt-get install sqlite3
```

但如果要基于 **SQLite3** 平台进行开发，这样做是不够的，因为它并不会安装 **SQLite3** 的源码和头文件，并且安装的只是 **x86** 主机的版本，因此最好直接从源码编译安装。

用以下命令可以下载 **SQLite3** 的源码包：

```
wget http://www.sqlite.org/sqlite-amalgamation-3.5.9.tar.gz
```

SQLite3 的源码有好几种发布形式，这里下载的是将所有代码都放在一个源文件中的形式，版本是 **3.5.9**。

交叉编译 **SQLite3** 的过程如下，这里用的是 **arm-linux-gcc-3.3.6** 编译器：

```
../sqlite-amalgamation-3.5.9/configure --host=${TARGET} --prefix=${SYSROOT}/usr
make
sudo PATH=${PATH} make install
```

其中 **TARGET** 变量的值是 **arm-linux**，**SYSROOT** 变量的值则是主机上的目标机根文件系统所在的目录。**--prefix** 参数用来指定安装目录的前缀，按上述命令配置，则安装后的可执行文件在 **\${SYSROOT}/usr/bin** 目录下，共享库在 **\${SYSROOT}/usr/lib** 目录下。

如果要在主机上对 **SQLite3** 进行本地编译，则过程如下，这里用的是 **gcc-4.3.2** 编译器：

```
../sqlite-amalgamation-3.5.9/configure
```

```
make
sudo make install
```

默认的安装目录前缀是 `/usr/local`，也就是说，可执行文件装在 `/usr/local/bin` 目录下，共享库装在 `/usr/local/lib` 目录下。

21.3 SQLite3 编程接口

SQLite3 提供了 C 语言的编程接口，使用起来十分方便。本节将介绍其中几个最基本的接口，有了它们就可以对 SQLite3 数据库进行任何操作。

21.3.1 打开和关闭数据库

使用 SQLite3 接口操作数据库时必须先将数据库打开。打开数据库的接口函数原型如下：

```
int sqlite3_open(const char *filename, sqlite3 **ppDb);
```

其各个参数及返回值的含义解释如下。

- ◆ `filename`：要打开的数据库的文件名。
- ◆ `ppDb`：指向一个 (`sqlite3 *`) 型指针，用于返回数据库句柄。
- ◆ 返回值：SQLITE_OK 代表成功，否则为错误码。

数据库打开以后，参数 `ppDb` 指向的指针变量的值就是数据库的句柄。数据库句柄是一个 (`sqlite3 *`) 型的数据，它所指向的结构体保存着对数据库操作的上下文，具体内容无须接口使用者关心。句柄就代表一个打开的数据库，在对数据库进行操作时都要把句柄的值传入以指定是对哪个数据库进行操作。

返回错误码的含义可查询 SQLite3 源码中的 `sqlite3.h` 头文件，这里将常见的一些错误码列举如下：

```
#define SQLITE_OK           0   /* 操作成功 */
#define SQLITE_ERROR        1   /* SQL 语句错误或找不到数据库 */
#define SQLITE_PERM        3   /* 访问被拒绝 */
#define SQLITE_ABORT        4   /* 回调函数返回放弃 */
#define SQLITE_BUSY        5   /* 数据库文件被加锁（使用中） */
#define SQLITE_LOCKED       6   /* 数据库表被加锁 */
#define SQLITE_NOMEM        7   /* 内存分配失败 */
#define SQLITE_READONLY     8   /* 数据库是只读的 */
#define SQLITE_IOERR        10  /* 磁盘 IO 失败 */
#define SQLITE_CORRUPT      11  /* 数据库已损坏 */
#define SQLITE_FULL        13   /* 数据库已满，插入失败 */
#define SQLITE_CANTOPEN     14   /* 无法打开数据库文件 */
#define SQLITE_EMPTY        16   /* 数据库是空的 */
#define SQLITE_MISMATCH     20   /* 数据类型不匹配 */
#define SQLITE_NOTADB       26   /* 被打开的文件不是 SQLite3 数据库 */
```

这些错误码是所有 SQLite3 编程接口共同使用的，而不仅仅是 `sqlite3_open` 函数。

打开的数据库不再使用时可用下面的接口函数关闭：

```
int sqlite3_close(sqlite3 *pDb);
```

其参数及返回值的含义解释如下。

- ◆ pDb：要关闭的数据库的句柄。
- ◆ 返回值：SQLITE_OK 代表成功，否则为错误码。

21.3.2 执行 SQL 语句

SQLite3 提供了一个一般化的函数接口，可以对已打开的数据库执行任何 SQL 语句。这个函数的原型如下：

```
int sqlite3_exec(sqlite3 *pDb, const char *sql,
                 int (*callback)(void* context, int nArg, char** azArg, char** azCol),
                 void *context, char **errmsg);
```

其各个参数及返回值的含义解释如下。

- ◆ pDb：已打开的数据库的句柄。
- ◆ sql：字符串，内容是一条或多条 SQL 语句。
- ◆ callback：回调函数。
- ◆ context：传递给回调函数的第一个参数，可称为上下文参数。
- ◆ errmsg：指向一个字符串，其内容是对操作中发生的错误的文字描述。
- ◆ 返回值：SQLITE_OK 代表成功，否则为错误码。

回调函数的主要作用是：当进行查询操作时，每查询到一条记录都会调用一次回调函数，这条记录的内容通过回调函数的参数传入。回调函数的参数及返回值的含义解释如下。

- ◆ context：调用 sqlite3_exec 时传入的 context 指针。
- ◆ nArg：数据的个数。
- ◆ azArg：字符串数组，有 nArg 个元素，表示每个字段的值。
- ◆ azCol：字符串数组，有 nArg 个元素，表示每个字段的名称。
- ◆ 返回值：0 表示要继续查询，否则查询操作结束，对 sqlite3_exec 函数的调用将返回错误码 SQLITE_ABORT。



提示

即使记录中某个字段的类型为整数，它的值也是由 C 语言的字符串来代表的。

如果对数据库做其他操作，则回调函数一般不会被调用，可为 NULL。

上下文参数将被透明的传递到回调函数中，可作为调用者向回调函数传递数据的通道使用。因它是 (void *) 型，故可以直接传递任意类型的指针作为参数。

调用 sqlite3_exec 时提供的 errmsg 参数是一个字符指针变量的地址，通过它将得到一个字符串的首地址。显然，这个字符串的存储空间是在 sqlite3_exec 函数内动态分配得到的，因此使

用过后必须释放以免造成内存泄漏。鉴于外界并不知道 `sqlite3_exec` 函数内部是如何进行内存分配的, SQLite3 平台专门提供了一个函数用于释放这块内存, 其原型如下:

```
void sqlite3_free(void *p);
```

其中参数 `p` 指向要释放的内存。

21.3.3 查询数据库

使用 `sqlite3_exec` 函数就可以对数据库进行查询操作, 但它使用回调函数作为返回结果的手段, 有时不是很方便, 因此 SQLite3 也提供了一个专门的接口进行查询操作, 其函数原型如下:

```
int sqlite3_get_table(sqlite3 *pDb, const char *sql,
    char ***pResult, int *nrow, int *ncolumn,
    char **errmsg);
```

其各个参数的含义解释如下。

- ◆ `pDb`: 已打开的数据库的句柄。
- ◆ `sql`: 字符串, 内容是一条或多条 SQL 语句。
- ◆ `pResult`: 一个 (`char **`) 型变量的地址, 用于返回查询结果, 查询结果由字符串数组代表。
- ◆ `nrow`: 结果表的行数。
- ◆ `ncolumn`: 结果表的列数。
- ◆ `errmsg`: 指向一个字符串, 其内容是对操作中发生的错误的文字描述。

`sqlite3_get_table` 函数与 `sqlite3_exec` 函数相比, 少了回调函数及上下文两个参数, 多了 `pResult`, `nrow`, `ncolumn` 三个参数用于返回查询结果。查询结果由一个字符串数组表示, 因此它的首地址是 (`char **`) 型。这是一个一维的字符串数组, 但要以二维的方式去理解, 其中前 `ncolumn` 个字符串表示结果表的字段名, 随后的 `ncolumn` 个字符串则是第一条记录的各个字段的值, 依此类推。因为结果中多了一行表示字段名的字符串, 因此数组的元素个数实际上是 $(nrow+1)*ncolumn$ 个。

显然, `pResult` 所指向的字符串数组及各个字符串所占的空间都是在 `sqlite3_get_table` 函数内部动态分配得到的, 因此使用后需要释放。由于它不是一块平坦的内存, 故也不能用 `sqlite3_free` 函数来释放, 而必须用下面这个专用的函数:

```
void sqlite3_free_table(char **result);
```

其中 `result` 参数就是查询得到的结果表的首地址。

21.4 使用 SQLite3 工具

SQLite3 平台还提供了一个命令行界面的交互式工具 `sqlite3`, 可以用它对 SQLite3 数据库进行操作。用如下命令可启动 `sqlite3` 工具:

```
sqlite3 student.db
```

其中 `student.db` 是要操作的数据库文件，如果它不存在则会自动创建一个新的空数据库。`sqlite3` 启动后将显示如下提示符：

```
sqlite>
```

这时可以输入 `SQL` 语句对数据库进行操作。语句可以换行，但必须以分号作为结束符，否则无法判断语句是否结束。

`sqlite3` 工具本身也支持一些命令可以操作数据库，为了与 `SQL` 语句相区别，它们都以小数点 `.` 开头。例如，在 `sqlite3` 界面中输入以下命令可显示所有命令的帮助信息：

```
.help
```

用如下的命令可以执行预先写在文件内的 `SQL` 语句：

```
.read test.sql
```

其中 `test.sql` 是文件名，它的内容必须是一条或多条 `SQL` 语句。

用如下的命令可以将文本文件中的内容导入到数据库的表中：

```
.import data.txt student
```

其中 `student` 是表的名称，而 `data.txt` 是一个文本文件，包含要导入的数据。数据的格式举例如下：

```
1|Zhang|M|18
2|Wang|F|17
```

这里每一行就是一条记录，竖线 `|` 分隔开的各个部分就是每个字段的值，如果分隔符不是竖线，则在导入前需要用以下命令修改默认的分隔符：

```
.separator ,
```

这里将分隔符修改成了逗号，这样，`sqlite3` 就能够识别逗号隔开的數據了。

用如下命令可将数据库的内容以 `SQL` 语句的形式全部显示出来：

```
.dump
```

退出 `sqlite3` 工具则使用下面的命令：

```
.quit
```

或者：

```
.q
```

21.5 SQLite3 数据库应用实例

在这一节里，我们将使用 `SQLite3` 平台编程对数据库进行查询操作。被查询的数据库的内容

用如下 SQL 语句建立:

```
create table Student (
    ID        int(5),
    Name      char(15),
    Sex       char(1),
    Age       int(3),
    primary key(ID)
);

create table Score (
    ID        int(5),
    Subject   char(10),
    score     int(3),
    primary key(ID, Subject)
);

insert into Student values(1, 'Zhang', 'M', 18);
insert into Student values(2, 'Wang', 'F', 17);
insert into Student values(3, 'Li', 'M', 19);

insert into Score values(1, 'English', 70);
insert into Score values(1, 'Math', 59);
insert into Score values(2, 'Math', 75);
```

21.5.1 使用 sqlite3_exec 查询数据库

例程源码如下:

```
/* 文件名: query.c */
/* 说明: 查询数据库例程, 使用 sqlite3_exec */

#include <stdio.h>
#include <sqlite3.h>

static char sql[] = "select * from student;"; /* 查询所用 SQL 语句 */

/* 回调函数 */
int my_callback(void *context, int num, char **pArg, char **pCol)
{
    int i;
    /* 输出记录中的每个字段 */
    for (i = 0; i < num; i++) {
        printf("%5s = %5s, ", pCol[i], pArg[i]);
    }
    printf("\n");
    return 0;
}

/* 主函数 */
int main(void)
```



```
{
    int err;
    sqlite3 *db;
    char *errmsg;
    /* 打开数据库 */
    err = sqlite3_open("student.db", &db);
    if (err != SQLITE_OK) {
        fprintf(stderr, "main: sqlite3_open(), %d\n", err);
        return -1;
    }
    /* 执行 SQL 语句 */
    err = sqlite3_exec(db, sql, my_callback, NULL, &errmsg);
    if (err != SQLITE_OK) {
        fprintf(stderr, "main: sqlite3_exec(), %d\n", err);
    }
    /* 释放错误信息所占的内存空间 */
    sqlite3_free(errmsg);
    /* 关闭数据库 */
    sqlite3_close(db);
    return 0;
}
```

编译这个例程的时候要注意与 `sqlite3` 共享库链接, 如:

```
gcc -Wall query.c -lsqlite3
```

例程的运行结果如下:

ID =	1,	Name =	Zhang,	Sex =	M,	Age =	18,
ID =	2,	Name =	Wang,	Sex =	F,	Age =	17,
ID =	3,	Name =	Li,	Sex =	M,	Age =	19,

21.5.2 使用 `sqlite3_get_table` 查询数据库

例程源码如下:

```
/* 文件名: query1.c */
/* 说明: 查询数据库例程, 使用 sqlite3_get_table */

#include <stdio.h>
#include <sqlite3.h>

static char sql[] = "select * from student;"; /* 查询所用 SQL 语句 */

/* 主函数 */
int main(void)
{
    int err;
    sqlite3 *db;
    char *errmsg;
    char **result;
```




```

int row, col;
int i, j;
/* 打开数据库 */
err = sqlite3_open("student.db", &db);
if (err != SQLITE_OK) {
    fprintf(stderr, "main: sqlite3_open(), %d\n", err);
    return -1;
}
/* 执行 SQL 语句 */
err = sqlite3_get_table(db, sql, &result, &row, &col, &errmsg);
if (err != SQLITE_OK) {
    fprintf(stderr, "main: sqlite3_exec(), %d\n", err);
}
/* 输出字段名 */
for (j = 0; j < col; j++) printf("%10s", result[j]);
printf("\n");
/* 输出各条记录 */
for (i = 1; i < row+1; i++) {
    for (j = 0; j < col; j++) printf("%10s", result[i*col+j]);
    printf("\n");
}
/* 释放错误信息所占的内存空间 */
sqlite3_free(errmsg);
/* 释放结果表所占的内存空间 */
sqlite3_free_table(result);
/* 关闭数据库 */
sqlite3_close(db);
return 0;
}

```

可以看出, 使用 `sqlite3_get_table` 函数查询数据库时不需要回调函数, 处理起来比较简单, 但也带来一个问题: 如果查询到的结果很多, 它们将同时消耗内存, 需要的内存容量将会很大。

例程的运行结果如下:

ID	Name	Sex	Age
1	Zhang	M	18
2	Wang	F	17
3	Li	M	19



第 22 章 产品开发实例：无线信息终端

在本书前面的章节中，主要讲述了嵌入式产品开发过程中使用到的各种技术。本章将通过一个典型嵌入式产品的开发，从硬件平台的选型到软件的设计，以及应用部署方案等各个方面介绍其实现。这样做的主要目的是希望读者能够从中借鉴一些嵌入式产品开发的经验。

要完整地说明一个复杂的终端产品的实现已经超出了本书的范围，因此本章在叙述时，只对软件的关键部分进行了较详细的说明，其余内容则做了简化。

22.1 总体架构

整个系统由无线信息终端与系统服务平台构成，如图 22.1 所示。各个无线信息终端通过有线或无线网络的方式连接到一个统一的业务接入平台。业务接入平台将来自终端的业务请求分发到各个应用服务器，并将应用服务器的回应转发给无线信息终端。

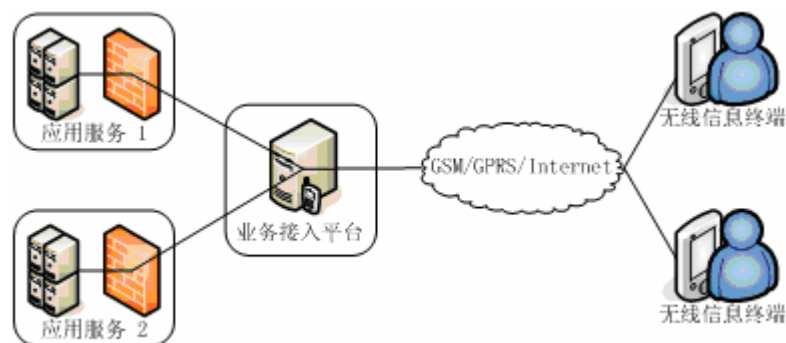


图 22.1 系统总体架构

应用服务就是用于支持终端的各种业务请求、提供具体的行业应用服务的系统，其硬件平台采用高档 PC 服务器或其他小型机系统，软件上则可以针对不同的行业应用提供差异化的行业应用服务软件。无线信息终端与业务接入平台之间的通信通过 IP 协议进行，但具体的底层可以是 GPRS，也可以是有线网络，与服务器之间的通信报文采用 HTTP 协议的格式。

本章所描述的产品即属于其中无线信息终端的部分。对于这种信息终端，它与通用的消费电子终端（如手机等）的显著不同在于，它搭载有专用的行业应用软件，使用这些行业应用软件的信息终端可称为行业应用终端。本章将以一款商业用零售渠道管理信息终端为例来说明整个终端产品的开发。如表 22.1 所示是所描述行业应用的总体要求，以功能菜单的形式给出。

表 22.1 零售渠道管理信息终端的功能菜单

一级菜单	二级菜单	功能说明
需求业务	月度需求	输入零售商品月度需求量，提交公司
	历史需求	查询以往月份零售需求量和评审量
	需求图表	查看需求量和评审量对比图
订货业务	商品订购	输入商品订购量，向公司提交订单
	订单查询	查询订单中商品信息明细
零售 POS	商品零售	零售户售卖商品，提供单据打印功能
	零售记录	记录销售明细和利润
	数据上报	上传销售信息到公司
	库存盘点	盘点零售户商品库存
	价格维护	修改商品价格信息
	销售反冲	商品退货处理
业务信息	电子结算	查询电子交易信息
	商品识别	识别商品
	违规查询	查看公司下发的违规信息
	通知查看	查看公司下发的通告
	政策法规	查看公司下发的政策
基础数据	商品信息	商品的详细信息查询
	零售户信息	查看零售户信息
移动业务	终端设置	修改终端设置
投诉建议	无	下载公司投诉模板
问卷调查	无	下载公司下发的问卷模板

22.2 硬件设计

终端的硬件平台是在 HY2410A 开发板的基础上设计的。当然，实际的产品不可能直接采用 HY2410A 开发板，而是在其基础上进行剪裁，增加很多必要的外部设备，如读卡器、打印机等，并根据产品的外壳重新设计制板。

根据总体方案，本终端的核心功能是零售渠道管理信息专用终端（以下简称零售信息终端），用于实现零售行业管理应用业务。在考虑终端系统硬件功能规范时，既要考虑应用业务的要求，也要考虑成本效益。因此，综合考虑性能和成本，最终制定终端的硬件系统功能规范如表 22.2 所示。

表 22.2 零售信息终端硬件功能规范

项目名称	性能参数
处理器	32 位 ARM9，主频 200MHz
操作系统	Linux 2.6 内核
存储	DRAM 64MB，NAND Flash 64MB

(续表)

项目名称	性能参数
显示屏	4.3 英寸 RGB 24bit 彩屏,分辨率 480 像素 x 272 像素,点距 0.198mm
图形界面	主流图形应用界面
触摸屏功能	支持
汉字输入	支持汉字输入
汉字显示	支持汉字显示
支持外设	键盘
通信方式	RS232, 以太网
USB 接口	Host 接口 1 个
工作电源	DC 9V/4A
功率	≤ 5W
工作温度	-10℃ ~ +55℃
湿度	0 ~ 90%

根据终端功能规范,我们选择性价比优异的 ARM 系列芯片。对于终端所需的应用来说,选择 S3C2410A 作为处理器已经足够。终端可以用于有线网络的环境,因此需要扩展以太网接口,这方面选择了性价比好、通信稳定的 CS8900A 芯片。另外很重要的一点,终端要求可以使用 GPRS 连接网络,因此需要扩展 GPRS 模块,这方面选择了性价比较好的 SIM-300 模块。

22.3 软件设计

整个终端软件包括系统软件与应用软件两大部分。系统软件涉及到操作系统的选择及相关驱动的开发。而应用软件则包括嵌入式图形系统的选择,它在很大程度上将决定终端软件的开发方式以及开发语言的选择。

在实际开发中,系统软件选择 Linux 2.6.30 作为操作系统内核,并搭配装有 busybox 1.10.2 的文件系统。而嵌入式图形系统则选用了 Qt2 平台。这里需要说明的是,虽然目前最新的 Qt 版本是 4.5.2,但考虑到终端的存储空间以及用户界面的响应速度,我们最终选用了较早的 Qt2 版本作为图形系统。Qt2 的编程接口与 Qt4 大体来说是类似的。

22.3.1 总体框架

终端软件的总体框架如图 22.2 所示。
下面对软件框架中的各个部分进行说明。

一、内核及驱动

系统采用 Linux 2.6.30 内核,将其移植到 S3C2410A 平台上。根据终端的功能,需要开发或移植的主要驱动有:键盘驱动、显示器驱动、触摸屏驱动。

二、图形界面及数据库平台

使用嵌入式 Qt 2.3.7 作为图形界面的开发平台,使用 SQLite3 作为数据库的开发平台。



图 22.2 终端软件总体框架

三、通信

终端与服务器间使用 IP 协议通信。使用串口控制 GPRS 通信模块接入网络，应用程序使用 TCP Socket 进行编程。

四、业务应用协议

业务应用协议封装为 HTTP 报文的格式，由其包体数据承载业务应用协议包。报文结构如图 22.3 所示。

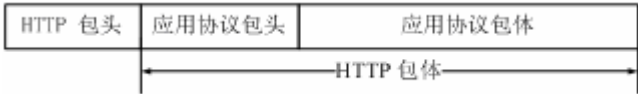


图 22.3 协议报文结构

终端应用软件采用请求/应答的同步通信模型与服务器进行通信，也就是说，终端在需要从服务器获取数据时，将发送一个请求。服务器处理完业务后，返回响应数据并断开连接。

终端与服务器间的业务应用协议需要根据业务要求进行设计。在这里，HTTP 协议只是用于终端与服务器间进行通信的底层协议，在 HTTP 包体中承载的具体业务应用协议则是根据业务的具体要求而设计的，也分为两个部分，即包头与包体。包头部分是固定长度的，而包体部分的长度可根据包头中的长度字段确定。

22.3.2 协议报文格式

22.3.2.1 HTTP 头的格式

采用 HTTP 1.1 协议的报文头，规定其中的必选字段如表 22.3 所示。

表 22.3 HTTP 头中的必选字段

字段	类型	说明
Post	String	Post 方式
User-Agent	String	服务端用做验证，以防止从网页中非法访问，取值：cjhy

(续表)

字段	类型	说明
Host	String	服务器地址
Content-Length	String	包体的长度
Content-Type	String	内容格式

举个例子如下：

```
POST / HTTP/1.1
User-Agent: cjhhy
Host: 192.168.1.21:8080
Content-Length: 322
Content-Type: text/plain
```

22.3.2.2 业务应用协议包头

业务应用协议包头中的各个字段如表 22.4 所示。

表 22.4 业务应用协议包头中的字段

字段名	长度	类型	说明
Length	8	Integer	表示消息包的长度
Term_ver	10	String	终端版本号
Manu_no	2	String	厂商代码
Oper_no	2	String	业务码
Func_no	2	String	操作码
Client_code	15	String	客户代码
Sim_no	20	String	SIM 卡号
Ret_type	2	String	返回码类型
Ret_code	2	String	返回码

其中 Integer 类型的数据是用其十进制形式的字符串表示的，这样做的好处是可以将整个报文作为字符串处理。如果报文中直接出现二进制数据，则有可能出现特殊字符以及字符串结束标志。

终端版本号、厂商代码、客户代码存放在本地数据库中，在业务使用期间是固定不变的。

业务码的取值对于本类终端来说固定为 90。

操作码是决定业务功能的一个重要字段，一些取值举例如表 22.5 所示。

表 22.5 操作码取值

功能	取值
终端版本认证	01
终端程序更新	02
服务器地址更新	03
商品品种更新	04
签到、签退	18
零售户信息查询	19

返回码类型有两种，00 表示是接入平台的返回码，10 表示是业务平台的返回码。

返回码表示操作是否成功，以及失败的原因。接入平台的返回码举例如表 22.6 所示。业务平台的返回码举例如表 22.7 所示。

表 22.6 接入平台返回码取值

返回码	含义
00	成功返回
01	已请求
02	非法终端（User-Agent 验证失败）
03	无效终端
04	无访问权限
05	内部错误

表 22.7 业务平台返回码取值

返回码	含义
00	成功返回
01	无须更新
02	业务繁忙，返回失败
03	不是合法客户
04	数据上传不成功
05	数据没找到
06	请求报文不完整
07	报文字段类型错误

对于终端请求报文来说，返回码类型和返回码都固定为 00。

22.4 应用软件

终端应用软件整体结构上划分为三个模块：GUI 应用模块、业务功能处理模块及通信协议模块。三者间的关系是 GUI 应用模块调用业务功能处理模块中的函数，而业务功能处理模块中调用通信协议模块中的函数。

本节将对软件中关键部分的实现举例进行说明。需要特别说明的是，本节中的实现很多地方做了简化处理。

22.4.1 GUI 应用模块

终端应用软件的开发是以 GUI 为中心的，这体现在如下两个方面。

- ◆ 一般应用的需求与设计是以 GUI 为导向的，最后软件的使用是围绕 GUI 的。
- ◆ GUI 代码量一般是最大的，而且变化也是最频繁的，维护工作繁重，因此这个部分的框架设计与代码管理非常重要。

因此在整个软件的设计中以 GUI 应用模块为中心定义与其他模块的分层接口。为了设计简洁，对于应用中出现的每个单独界面设计了一个窗口，这样每个窗口的代码就自然形成了一个子模块，如图 22.4 所示是 GUI 应用模块中部分窗口子模块。

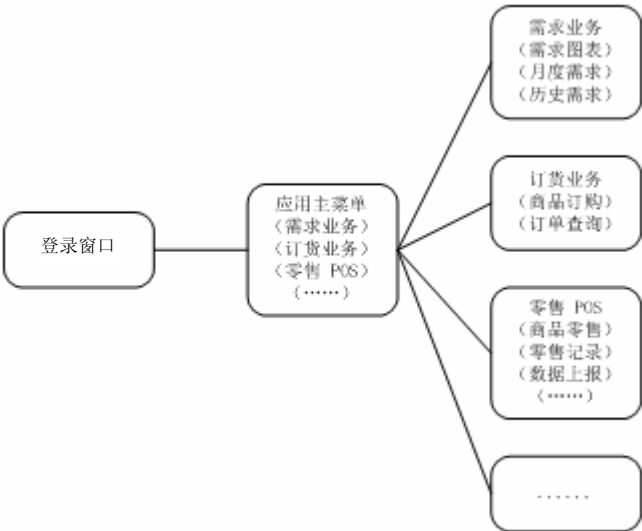


图 22.4 GUI 应用模块中的窗口子模块

最终界面的图像举例如图 22.5（登录窗口）、图 22.6（主菜单）及图 22.7（需求业务界面）所示。

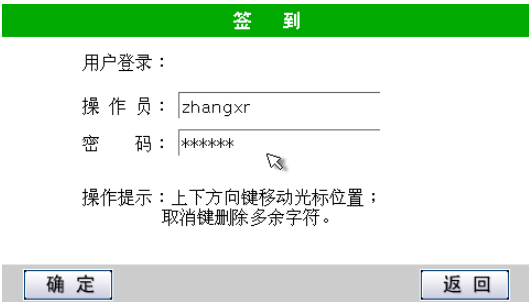


图 22.5 登录界面



图 22.6 一级菜单界面



图 22.7 二级菜单界面

登录界面使用 Qt 的 UI 设计器 designer 进行设计。而主菜单界面与二级菜单界面具有相似的外观布局，界面元素比较少且位置有规律，因此可采用统一的代码以生成界面。这样做还有以下好处。

- ◆ 一般的智能信息终端具有类似的界面，因此代码可以很容易地借鉴到其他项目中。
- ◆ 所有界面的风格、图标的位置一致。可以想象，如果每个界面都用 designer 进行设计，则为了让它们的按钮位置相同就需要负出很大工作量。
- ◆ 具有较高的灵活性，如果界面需要调整，则只需修改一份代码即可。

主界面窗口类的定义如下：

```
// 文件名: zzjm.h
// 说明: 主菜单界面

#ifndef ZZJM_INCLUDED
#define ZZJM_INCLUDED

#include <qwidget.h>
#include "yeqwidget.h"

class zzjmForm : public QWidget, YeqWidget
{
    Q_OBJECT
public:
    zzjmForm(QWidget* parent = 0, const char* name = 0);
    virtual ~zzjmForm();
public:
    static zzjmForm *ptr; // 指向本类唯一实例的静态指针
protected slots:
    virtual void xuqiuyewu(); // 需求业务
    virtual void dinghuoyewu(); // 订货业务
    virtual void lingshoupos(); // 零售 POS
    virtual void yewuxinxi(); // 业务需求
    virtual void jichushuju(); // 基础数据
    virtual void yidongyewu(); // 移动业务
    virtual void tousujianyi(); // 投诉建议
    virtual void wenjuandiaochoa(); // 问卷调查
    virtual void fanhui(); // 返回
```

```
protected:
    void customEvent(QCustomEvent*);
};

#endif
```

其中 `zzjmForm` 是代表主菜单界面的窗口，因此要从 `QWidget` 继承，而 `YeqWidget` 类则是实现前述代码生成界面的一个辅助类。在类中，对应于界面上的每一个按钮定义了一个相应的槽。此外，类似于这样的每个界面一般都只会生成一个唯一的实例，因此定义了一个静态指针指向这个唯一实例，在其他类中就可以用 `zzjmForm::ptr` 这种形式访问这个对象，使用起来比较方便。

`zzjmForm` 类的构造函数如下：

```
zzjmForm::zzjmForm(QWidget* parent, const char* name)
: QWidget(parent, name, Qt::WStyle_Customize|Qt::WStyle_NoBorder)
, YeqWidget(this)
{
    ptr = this; // 静态指针指向自己
    setWidget("bar/ywgn.png"); // 增加标题栏
    addFootButtonPixmap(3, "foot/fh.png",
        Qt::Key_Escape, SLOT(fanhui())); // 增加返回按钮
    addButton(0, "btn/xqyw.png", "misc/1.png",
        Qt::Key_1, SLOT(xuqiuyewu()), FALSE); // 增加需求业务按钮
    addButton(1, "btn/dhyw.png", "misc/2.png",
        Qt::Key_2, SLOT(dinghuoyewu()), FALSE); // 增加订货业务按钮
    addButton(2, "btn/lspw.png", "misc/3.png",
        Qt::Key_3, SLOT(lingshoupos()), FALSE); // 增加零售 POS 按钮
    addButton(3, "btn/ywxx.png", "misc/4.png",
        Qt::Key_4, SLOT(yewuxinxi()), FALSE); // 增加业务信息按钮
    addButton(4, "btn/jcsj.png", "misc/5.png",
        Qt::Key_5, SLOT(jichushuju()), FALSE); // 增加基础数据按钮
    addButton(5, "btn/ydyw.png", "misc/6.png",
        Qt::Key_6, SLOT(yidongyewu()), FALSE); // 增加移动业务按钮
    addButton(6, "btn/tsjy.png", "misc/7.png",
        Qt::Key_7, SLOT(tousujianyi()), FALSE); // 增加投诉建议按钮
    addButton(7, "btn/wjdc.png", "misc/8.png",
        Qt::Key_8, SLOT(wenjuandiaocha()), FALSE); // 增加问卷调查按钮
    installEventFilter(this); // 安装过滤器
}
```

在界面实现上，我们没有使用 Qt 窗口的标题栏，而是将窗口显示为无边框的，然后在顶部增加一个标签来模拟标题栏。其中的 `setWidget`, `addFootButtonPixmap`, `addButton` 等函数是前述 `YeqWidget` 类的成员，用来为窗口增加各种部件，并设置快捷键，连接信号与槽。在这些标签和按钮中，使用了大量的 PNG 图片。

以需求业务为例，其对应的槽的代码如下：

```
void zzjmForm::xuqiuyewu()
{
    if (xqywForm::ptr == NULL)
    {
```

```

        new xqywForm(0, "xqywGui");
    }
    xqywForm::ptr->show();
}

```

在这里只需要显示二级菜单——需求业务界面，如果这个界面的实例还不存在，则先创建一个对象。这里，使用 `new` 操作符得到的指针不需要保存，因为新对象的指针自动保存在它的静态成员 `ptr` 中。需求业务界面的实现则与主菜单界面类似。

在二级菜单以下具体业务流程的界面各不相同，因此可以回到用 `designer` 设计界面的方式。

有了这样一个 GUI 代码框架后，可以根据业务的要求，陆续将其他界面加入，从而逐步成为一个大的工程。要特别说明的是，一个好的工程管理对一个项目是非常重要的。幸运的是，Qt2 也提供了用于管理工程的工具 `progen` 和 `tmake`，`progen` 用于生成工程文件，`tmake` 用于生成 `Makefile`。

在开发过程中，很多调试工作可以在主机上进行，但有时又要在目标机上进行测试，因此经常需要从一种编译环境切换到另一种编译环境。下面给出一段用于设置环境变量的脚本：

```

TARGET=arm-linux
# QT 2.3.7 & Qtopia 1.7.0 settings
TMAKEDIR=/opt/tmake-1.13
if [ "${TARGET}" = "arm-linux" ]; then
    TMAKEPATH=${TMAKEDIR}/lib/qws/linux-arm-g++
    QTDIR=${SYSROOT}/opt/qt-2.3.7
    QPEDIR=${SYSROOT}/opt/qtopia-free-1.7.0
else
    TMAKEPATH=${TMAKEDIR}/lib/qws/linux-generic-g++
    QTDIR=/opt/qt-2.3.7
    QPEDIR=/opt/qtopia-free-1.7.0
fi
PATH=${QTDIR}/bin:${TMAKEDIR}/bin:${PATH}
LD_LIBRARY_PATH=${QTDIR}/lib:${QPEDIR}/lib
export TMAKEDIR TMAKEPATH QTDIR QPEDIR PATH LD_LIBRARY_PATH

```

在上述脚本中，如果将第一行的变量定义去掉，则其他环境变量将定义为本地开发环境所需的，否则就定义为 ARM 开发环境所需的。

22.4.2 通信协议模块

通信协议模块处于三个模块中的最底层位置，由业务功能模块调用。它的主要功能如下。

- ◆ 将上层的业务应用报文包装成 HTTP 的格式发送给服务器。
- ◆ 将从服务器接收到的信息解析，去掉 HTTP 报文头后传递给上层。

本模块只处理通信和 HTTP 包头的解析，具体的业务应用报文则在业务功能模块中进行解析。它主要向上层业务功能模块提供通信接口，定义如下：

```

int send2server(char *iMsg, char **oMsg, int *len);

```

其各个参数及返回值的含义解释如下。

- ◆ iMsg: 要发送给服务器的业务应用报文。
- ◆ oMsg: 指向一个字符指针, 用于返回接收到的业务应用报文。
- ◆ len: 指向一个整型数, 用于返回包的长度。
- ◆ 返回值: 0 表示成功, -1 表示有错误发生。

在 send2server 函数中将完成 HTTP 协议的编解码功能。当然, 我们只需要实现项目中协议所规定的功能, 并不需要完整实现 HTTP 协议。

函数的实现如下:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUF_LEN      1024
#define SZ_LINE_BUF 50

extern char *yc_ipaddr; /* 服务器地址 */
extern int  yc_port;    /* 服务器端口 */

int send_byte(int sock, const char *buf, int len);
int recv_byte(int sock, char *buf, int len);
int tcps_recv_http(int sock, char *buf, int len,
                   int *body_start, int *body_len);
int tcps_recv_http_header(int sock, char *buf, int len,
                          int *body_start, int *body_len);

int send2server(char *iMsg, char **oMsg, int *len)
{
    int r;
    int clifd;
    struct sockaddr_in servaddr;
    socklen_t socklen;
    struct timeval tv;
    char *buf, *p;
    int body_start;
    /* 分配缓冲区空间 */
    buf = malloc(BUF_LEN);
    if (buf == NULL) {
        printf("not enough memory!\n");
        return -1;
    }
    /* 向缓冲区中写入 HTTP 头 */
```

```

p = buf;
p += sprintf(p, "POST / HTTP/1.1\r\n");
p += sprintf(p, "User-Agent: cjhy\r\n");
p += sprintf(p, "Host: %s:%d\r\n", yc_ipaddr, yc_port);
p += sprintf(p, "Content-Length: %d\r\n", strlen(iMsg));
p += sprintf(p, "Content-Type: text/plain\r\n");
p += sprintf(p, "\r\n\r\n");
/* 写入上层报文 */
strcpy(p, iMsg);
/* 打开 socket */
if ((clifd = socket(PF_INET, SOCK_STREAM, 0)) < 0) {
    free(buf);
    printf("create socket error!\n");
    return -1;
}
/* 设置服务器地址 */
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
inet_aton(yc_ipaddr, &servaddr.sin_addr);
servaddr.sin_port = htons(yc_port);
/* 设置超时 */
tv.tv_sec = 30;
tv.tv_usec = 0;
setsockopt(clifd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));
setsockopt(clifd, SOL_SOCKET, SO_SNDTIMEO, &tv, sizeof(tv));
/* 连接 */
socklen = sizeof(servaddr);
if (connect(clifd, (struct sockaddr *)&servaddr, socklen) < 0) {
    close(clifd);
    free(buf);
    printf("can't connect to %s!\n", yc_ipaddr);
    return -1;
}
/* 发送 */
r = send_byte(clifd, buf, strlen(buf));
if (r < 0) {
    close(clifd);
    free(buf);
    printf("send fail!\n");
    return -1;
}
/* 接收服务器返回的信息 */
r = tcps_recv_http(clifd, buf, BUF_LEN, &body_start, len);
if (r < 0) {
    close(clifd);
    free(buf);
    printf("error when recieve data from server!");
    return -1;
}
/* 分配输出缓冲区并将 HTTP 包体复制过去 */
*oMsg = malloc(*len+1);

```



```
memcpy(*oMsg, buf+body_start, *len);
(*oMsg)[*len] = '\\0'; /* 写结束符 */
close(clifd);
free(buf);
return 0;
}

/* 可靠发送 len 个字节 */
int send_byte(int sock, const char *buf, int len)
{
    int rc;
    int byte;
    for (byte = 0; byte < len; byte += rc) {
        rc = send(sock, buf+byte, len-byte, MSG_NOSIGNAL);
        if (rc < 0 && errno != EINTR) {
            byte = -1;
            break;
        }
    }
    return byte;
}

/* 可靠接收 len 个字节 */
int recv_byte(int sock, char *buf, int len)
{
    int rc;
    int byte;
    for (byte = 0; byte < len; byte += rc) {
        rc = recv(sock, buf+byte, len-byte, MSG_NOSIGNAL);
        if (rc == 0) break;
        if (rc < 0 && errno != EINTR) {
            byte = -1;
            break;
        }
    }
    return byte;
}

/* 接收 HTTP 包 */
int tcps_recv_http(int sock, char *buf, int len,
                  int *body_start, int *body_len)
{
    int byte, bytel;
    DBGP("tcps_recv_http: wait for receiving...\n");
    /* 先接收 HTTP 包头 */
    byte = tcps_recv_http_header(sock, buf, len, body_start, body_len);
    if (byte < 0) {
        ERRP("tcps_recv_http: recv_http_header FAILED!");
        return -1;
    }
    /* 再接收 HTTP 包体 */
}
```



```

    if (byte >= *body_start+*body_len) { /* 已经全部接收到 */
        DBGP("tcps_recv_http: data ready!\n");
    } else if (*body_start+*body_len > len) { /* 缓冲区太小 */
        DBGP("tcps_recv_http: buffer too small!\n");
    } else { /* 接收剩余部分 */
        byte1 = recv_byte(sock, buf+byte, *body_start+*body_len-byte);
        if (byte1 < 0) {
            ERRP("tcps_recv_http: recv_byte FAILED!");
            return -1;
        }
        byte += byte1;
    }
    DBGP("tcps_recv_http: %d bytes received! "
        "body start at %d, "
        "body len = %d\n", byte, *body_start, *body_len);
    DUMP("tcps_recv_http buf", buf, byte);
    return byte;
}

/* 接收 HTTP 包头 */
int tcps_recv_http_header(int sock, char *buf, int len,
    int *body_start, int *body_len)
{
    int i;
    int rc;
    int byte;
    char *p1, *p2;
    char line_buf[SZ_LINE_BUF+1];
    int line_start;
    int line_len;
    int header_end = 0;
    line_start = 0;
    *body_len = 0;
    /* 循环接收直到包头结束 */
    for (byte = 0; !header_end && byte < len; byte += rc) {
        rc = recv(sock, buf+byte, len-byte, MSG_NOSIGNAL);
        if (rc == 0) break;
        if (rc < 0 && errno != EINTR) {
            byte = -1;
            break;
        }
    }
    for (i = byte; i < byte+rc; i++) {
        /* 不是新行则继续 */
        if (buf[i] != '\n' || i < 1 || buf[i-1] != '\r') continue;
        /* 将新行复制到 line_buf 中 */
        line_len = MIN(i-1-line_start, SZ_LINE_BUF);
        memcpy(line_buf, buf+line_start, line_len);
        line_buf[line_len] = 0;
        /* 分析这一行 */
        for (p1 = line_buf; *p1 == ' ' && *p1 != 0; p1++);
        /* 如果是空行说明包头结束 */
    }
}

```



```

        if (*p1 == 0) {
            *body_start = i+1;
            header_end = 1;
            break;
        }
        /* 分析字段 Content-Length, 得到包体长度 */
        p2 = strchr(p1, ':');
        if (p2 != NULL) {
            *p2 = 0;
            for (p2++; *p2 == ' ' && *p2 != 0; p2++);
            DBGP("tcps_rcv_http_header: %s = %s\n", p1, p2);
            if (strcmp(p1, "Content-Length") == 0) {
                *body_len = atoi(p2);
            }
        } else {
            DBGP("tcps_rcv_http_header: %s\n", p1);
        }
        line_start = i+1;
    }
}

if (!header_end && byte == len) { /* 未接收完但缓冲区已满 */
    DBGP("tcps_rcv_http_header: buffer too small!\n");
    return -1;
}

return byte;
}

```

从中可以看到, 由参数 `oMsg` 返回的指针所指向的缓冲区是在函数内动态分配的, 因此需要在上层业务功能模块中释放, 以免造成内存泄漏。

用 `sprintf` 函数可以很方便地构造出所需的 HTTP 包头。对 HTTP 包头的解析则采用逐个字符分析的办法, 如果发现回车换行符, 则说明一行文本结束, 对这一行进行分析。如果发现一个空行, 说明 HTTP 包头结束, 后面的内容属于 HTTP 包体。实际上, 我们在接收时只需分析 `Content-Length` 数据域, 得到包体长度即可。

22.4.3 业务功能模块

业务功能模块处理业务应用协议, 从软件层次上看处于 GUI 应用模块下面, 而在通信协议模块之上。用户从界面上发起操作, 然后调用业务功能模块提供的功能接口, 如果需要与服务器交换数据, 则业务功能模块将按业务应用协议的格式构造报文, 递交给通信协议模块以发送给服务器, 并将服务器返回的应答报文进行解析, 最后将结果传递给 GUI 界面模块, 显示给用户。

下面以零售户信息查询功能为例来说明业务功能模块的实现。这项功能需要与服务器交换数据, 但所发送的报文中包体是空的, 也就是不需要上层提供其他参数, 因此定义接口的原型如下:

```
int LingQuery(char **OutMsg);
```

其中 `OutMsg` 指向一个字符指针, 用于返回业务应用协议包体数据。返回值是数据的长度, 如果操作失败则为负数。



函数的实现如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sqlite3.h>

#include <comm.h> /* 通信协议模块头文件 */

#define PACKHEADLEN 63

/* 构造业务应用协议包头 */
int PacketHead(char *Func_no, char *Sim_no, char *InMsg, char *OutMsg)
{
    char sql[128];
    char *errmsg;
    sqlite3 *db;
    char **dbResult;
    int rc, nRow, nColumn;
    char *p;
    int len;
    if (InMsg == NULL) return -1;
    len = strlen(InMsg)+PACKHEADLEN; /* 长度域包括包头的长度 */
    p = OutMsg;
    p += sprintf(p, "%08d", len); /* 消息包的长度 */
    /* 查询数据库，得到终端版本号、厂商代码、客户代码 */
    rc = sqlite3_open("/opt/cjhy.db", &db);
    if (rc != SQLITE_OK) {
        fprintf(stderr, "Can't open database\n");
        sqlite3_close(db);
        return -1;
    }
    printf("open database successfully!\n");
    sprintf(sql, "select programversion, termid, customid from terminfo;");
    rc = sqlite3_get_table(db, sql, &dbResult, &nRow, &nColumn, &errmsg);
    if (rc != SQLITE_OK) {
        fprintf(stderr, "Query database fail: %s\n", errmsg);
        sqlite3_free(errmsg);
        sqlite3_close(db);
        return -1;
    }
    sqlite3_free(errmsg);
    p += sprintf(p, dbResult[nColumn]); /* 终端版本 */
    p += sprintf(p, dbResult[nColumn+1]); /* 厂商代码 */
    p += sprintf(p, "90"); /* 业务码 */
    p += sprintf(p, Func_no); /* 操作码 */
    p += sprintf(p, dbResult[nColumn+2]); /* 客户代码 */
    sqlite3_free_table(dbResult);
    sqlite3_close(db);
}
```

```

    p += sprintf(p, Sim_no); /* SIM 卡号 */
    p += sprintf(p, "00"); /* 返回码类型 */
    p += sprintf(p, "00"); /* 返回码 */
    strcpy(p, InMsg);
    return 0;
}

/* 零售户信息查询 */
int LingQuery(char **OutMsg)
{
    char SendBuf[128];
    char *ReadBuf;
    int iRet;
    int rlen = 0;
    char Ret_type[3], Ret_code[3];
    /* 加上包头, 构造业务应用协议包 */
    /* 报文体为空 */
    iRet = PacketHead("19", "00000000000000000001", "", SendBuf);
    if (iRet < 0) return -1;
    iRet = send2server(SendBuf, &ReadBuf, &rlen);
    if (iRet <= 0) return -1;
    memcpy(Ret_type, ReadBuf+PACKHEADLEN-4, 2); /* 取返回码类型 */
    Ret_type[2] = '\0';
    memcpy(Ret_code, ReadBuf+PACKHEADLEN-2, 2); /* 取返回码 */
    Ret_code[2] = '\0';
    if ((strcmp(Ret_type, "00") == 0) && (strcmp(Ret_code, "00") != 0)) {
        /* 接入平台返回失败, 复制错误码到输出缓冲区 */
        *OutMsg = malloc(5);
        memcpy(*OutMsg, ReadBuf+PACKHEADLEN-4, 4);
        (*OutMsg)[5] = '\0';
        free(ReadBuf);
        return -2;
    }
    if ((strcmp(Ret_type, "10") == 0) && (strcmp(Ret_code, "00") != 0)) {
        /* 业务平台返回失败, 复制错误码到输出缓冲区 */
        *OutMsg = malloc(5);
        memcpy(*OutMsg, ReadBuf+PACKHEADLEN-4, 4);
        (*OutMsg)[5] = '\0';
        free(ReadBuf);
        return -2;
    }
    /* 返回包体 */
    rlen -= PACKHEADLEN;
    *OutMsg = malloc(rlen);
    strcpy(*OutMsg, ReadBuf+PACKHEADLEN, rlen);
    free(ReadBuf);
    return rlen;
}

```

22.4.4 使用多线程读取设备

在上述终端软件的实现中，有多个外部设备需要操作。这些设备均可以设备文件的形式访问。一般来说，访问设备文件时，使用阻塞方式比较合适，这样就不必对设备进行轮询操作了。

但是对于有用户界面的软件来说，这样会带来一个问题，即如果进程阻塞在对设备文件的操作上，则界面将失去响应。这时可以考虑多线程的工作方式，让一个线程专门负责操作设备文件，当读到数据时通过某种方式通知用户界面线程。

Qt 本身支持多线程。使用 Qt 线程的好处是它与整个 Qt 平台更兼容。Qt 也支持异步发送事件的机制，可以用于线程间的通信。

具体实现示例如下：

```
#include <qapplication.h>
#include <qthread.h>
#include <qevent.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>

#include "qd.h" /* 签到界面 */

// 定义一个用户定制事件类型，用于包装接收到的设备数据
const QEvent::Type MyEvent = (QEvent::Type)1234;

// 继承 QThread 类
class GeneralDevThread : public QThread {
public:
    virtual void run();
};

// 读取设备文件的线程工作函数
void GeneralDevThread::run(void)
{
    int fd;
    char rx_buf[32];
    // 打开文件
    if ((fd = open("/dev/ttyS1", O_RDWR)) < 0) {
        perror("can not open device!\n");
        QThread::exit();
    }
    // 循环
    while (1) {
        if (read(fd, rx_buf, 32) < 0) {
            perror("read error!\n");
            QThread::exit();
        }
        // 向当前激活的窗口发送事件
        qApp->postEvent(qApp->activeWindow(),
```



```

        new QCustomEvent(MyEvent, rx_buf));
    // 唤醒主线程
    qApp->wakeUpGuiThread();
}
}

QApplication *app;

int main(int argc, char *argv[])
{
    app = new QApplication(argc, argv);
    app->setFont(QFont("wenquanyi", 16, 50, 0));
    // 安装翻译器
    QTranslator qt(0);
    qt.load("yc.qm", "/opt/");
    app->installTranslator(&qt);
    // 显示签到界面
    qdForm *loginwin = new qdForm(0, "LOGIN_WIN");
    app->setMainWidget(loginwin);
    loginwin->show();
    // 启动读设备线程
    GeneralDevThread devTh;
    devTh.start();
    // 进入主事件循环
    int r = app->exec();
    delete app;
    return r;
}

```

在接收事件的窗口中，可以覆盖其 `customEvent` 事件处理函数来实现对定制事件的处理，举例如下：

```

void zzjmForm::customEvent(QCustomEvent *event)
{
    if (event->type() == 1234) {
        int i;
        char *p = (char *)event->data();
        for (i = 0; i < 32; i++) printf("%x ", p[i]);
        printf("\n");
    } else {
        printf("unkown event\n");
    }
}

```

这里只是将读到的数据输出到了控制台上，实际上可以做更复杂的处理。

22.4.5 模块集成

在实际软件开发中，一个很重要的方法是模块化，将一个大的软件按功能划分为多个模块，模块与模块间定义相互调用的接口。原则上，模块能够单独进行开发测试。最后根据定义的接口集成



在一起形成整个软件。

在本项目中将整个软件大致划分为前述三个模块。可以看出，业务功能模块与通信协议模块都是使用 C 语言编写的，而 GUI 应用模块因为要用 Qt 平台，所以只能用 C++ 语言编写。最后集成时要考虑 C 与 C++ 混合编程的问题。

在调用接口函数时也要注意，在上述实例中多处出现被调用函数内部动态分配内存，然后返回给调用者的数据传递方式，因此要求调用者负责释放。



附录A 缩 略 语

ABI (Application Binary Interface, 应用程序二进制接口)
ADC (Analog to Digital Convertor, 模数转换器)
AHB (Advanced High performance Bus, 高性能总线)
AMBA (Advanced Microcontroller Bus Architecture, 高级微控制器总线架构)
API (Application Programming Interface, 应用编程接口)
ARM (Advanced RISC Machine, 高级 RISC 机器)
ASCII (American Standard Code for Information Interchange, 美国信息交换标准代码)
ATPCS (ARM/Thumb Procedure Call Standard, ARM/Thumb 过程调用标准)
CPSR (Current Program Status Register, 当前程序状态寄存器)
CPU (Central Processing Unit, 中央处理单元)
DMA (Direct Memory Access, 直接内存访问)
DCE (Data Circuit-terminating Equipment, 数据通信设备)
DHCP (Dynamic Host Configuration Protocol, 动态主机配置协议)
DTE (Data Terminal Equipment, 数据终端设备)
EABI (Embedded Application Binary Interface, 嵌入式应用程序二进制接口)
EIA (Electronic Industries Alliance, 电子工业协会)
FIFO (First In First Out, 先进先出)
FTP (File Transfer Protocol, 文件传输协议)
GID (Group ID, 组标识符)
GPIO (General-Purpose IO, 通用目的输入/输出)
HTML (Hyper Text Markup Language, 超文本标记语言)
HTTP (Hyper Text Transfer Protocol, 超文本传输协议)
I2C (Inter-Integrated Circuit, 集成电路间的一种通信协议)
I2S (Inter-IC Sound, 集成电路间音频)
IDE (Integrated Device Electronics, 集成设备电子联盟定义的硬盘接口标准)
IEEE (The Institute of Electrical and Electronics Engineers, 电气和电子工程师协会)
IP (Internet Protocol, 互联网协议)
IPC (Inter-process communication, 进程间通信)
ISO (International Organization for Standardization, 国际标准化组织)
JFFS (Journalling Flash File System, 日志型 Flash 文件系统)
MAC (Media Access Control, 媒体访问控制)
MBR (Main Boot Record, 主引导记录)
MMC (Multi-Media Card, 多媒体卡)

MMU (Memory Management Unit, 内存管理单元)
NPTL (Native POSIX Thread Library, 原生 POSIX 线程库)
OHCI (Open Host Controller Interface, 开放主机控制器接口)
PCB (Process Control Block, 进程控制块)
PCI (Peripheral Component Interface, 外围器件接口)
PCMCIA (Personal Computer Memory Card International Association, 个人计算机存储卡国际联合会定义的接口标准)
PID (Process ID, 进程标识符)
POSIX (Portable Operating System Interface, 可移植操作系统接口)
PWM (Pulse Width Modulation, 脉冲宽度调制)
RISC (Reduced Instruction Set Computer, 精简指令集计算机)
RTC (Real Time Clock, 实时时钟)
SD (Secure Digital Memory Card, 安全数字存储卡)
SMP (Symmetric Multiprocessor, 对称多处理器)
SoC (System on Chip, 片上系统)
SPI (Serial Peripheral Interface, 串行外设接口)
SPSR (Saved Program Status Register, 备份程序状态寄存器)
TCP (Transmission Control Protocol, 传输控制协议)
TGID (Thread Group ID, 线程组标识符)
TID (Thread ID, 线程标识符)
UART (Universal Asynchronous Receiver/Transmitter, 通用异步接收与发送器)
UDP (User Datagram Protocol, 用户数据报协议)
UID (User ID, 用户标识符)
UP (Uniprocessor, 单处理器)
URB (USB Request Block, USB 请求块)
USB (Universal Serial Bus, 通用串行总线)
XIP (Executing In Place, 原地执行)

◆ ◆ ◆

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：(010)88254396；(010)88258888

传 真：(010)88254397

E - mail：dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

电子工业出版社总编办公室

邮 编：100036